Titre: Title:	Implémentation matérielle d'un réseau sur puce et analyse du fonctionnement dans un environnement multiprocesseurs
Auteur: Author:	Francis St-Pierre
Date:	2006
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	St-Pierre, F. (2006). Implémentation matérielle d'un réseau sur puce et analyse du fonctionnement dans un environnement multiprocesseurs [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7833/

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

IMPLÉMENTATION MATÉRIELLE D'UN RÉSEAU SUR PUCE ET ANALYSE DU FONCTIONNEMENT DANS UN ENVIRONNEMENT MULTIPROCESSEURS

FRANCIS ST-PIERRE

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE EN SCIENCES APPLIQUÉES

(GÉNIE ÉLECTRIQUE)

NOVEMBRE 2006

Your file Votre référence
ISBN: 978-0-494-25577-3
Our file Notre référence
ISBN: 978-0-494-25577-3

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

IMPLÉMENTATION MATÉRIELLE D'UN RÉSEAU SUR PUCE ET ANALYSE DU FONCTIONNEMENT DANS UN ENVIRONNEMENT MULTIPROCESSEURS

Présenté par : ST-PIERRE Francis

En vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

Est évalué par le jury d'examen constitué de :

M. MARTEL Sylvain, Ph.D., Président de jury

M. BOIS Guy, Ph.D., directeur et membre

M. SAVARIA Yvon, Ph.D., co-directeur et membre

M. DAVID Jean Pierre, Ph.D., membre

REMERCIEMENTS

Je tiens d'abord à remercier mon directeur de recherche, Guy Bois. Son soutien moral constant m'a permis de mener à terme mon projet de recherche. Par ailleurs, je voudrais remercier mon co-directeur, Yvon Savaria, de même que Michel Langevin pour l'apport d'idées et de pistes à exploiter qu'ils ont pu me fournir tout au long du projet. Ainsi que Ami Castonguay pour son dévouement dans nos travaux de recherches conjoints.

Je ne voudrais surtout pas oublier mes chers camarades d'étude pour avoir partagé ce si beau temps à la maîtrise : François Deslauriers, Jean-François Thibeault, Patrick Samson, Simon Provost, Mortimer Hubin, Luc Filion, Jérôme Chevalier, Maxime De Nanclas ainsi que les autres membres du CIRCUS.

Salut à ma gang de fêtards du GRM : Bruno Tanguay, Jean-François Saheb, Serge Catudal, Ghislain Provost, Hung Tien Bui, Sébastien Régimbal, Mame Maria Mbaye, Guillaume Wild ainsi que tout le monde à l'association étudiante.

Un grand merci à Réjean Lepage et Alexandre Vesey pour leur soutien technique et bien plus. Sans oublier notre chère Ghyslaine qui a toujours été là pour s'occuper de nous.

Je souhaite terminer ces remerciements avec les personnes en qui j'ai le plus grand des respects. Mon ami Marc Létourneau, sans lui, je n'aurais pas étudié la microélectronique. Mon oncle Armand St-Pierre, pour son exemple dans le domaine des études universitaires. Une pensée spéciale pour ma grand-mère maternelle, Simone Courtois qui nous a quittés cette année. Finalement, mes parents en or, Gérard et Pauline ainsi que mon meilleur ami et plus jeune frère, Jérémie à qui je dédicace ce mémoire.

RÉSUMÉ

Dans le but de réduire la complexité des systèmes sur puce et ainsi d'en faciliter la conception, il est nécessaire d'identifier les aspects qui composent leur complexité. Diviser le système en plusieurs blocs est une des méthodes fréquemment utilisées. De tels blocs peuvent être des processeurs, des mémoires, des contrôleurs d'entrées et de sorties et des blocs matériels spécialisés. Pour que l'ensemble du système fonctionne, il est nécessaire que ces blocs soient connectés entre eux par un moyen de communication efficace. Pour ce faire, un nouveau paradigme est apparu pour répondre aux besoins des SoC (*Systems on Chip*), les réseaux intégrés sur puce (NoC). Pars ces réseaux la transmission de donnée se fait par paquets d'une source à sa destination. Les paquets sont routés dans le réseau selon un algorithme propre à chaque NoC. Plusieurs architectures et produits commerciaux existent, mais peu de détails sur ces réseaux et leur implémentation matérielle sont révélés. Ce mémoire passe en revue différentes architectures et présente un réseau intégré sur puce, le *rotator-on-chip* (RoC). Développée en partenariat avec STMicroelectronics de Ottawa, ce réseau est simple, extensible et est basé sur un modèle utilisant une topologie en anneau. Le RoC est configurable selon différents paramètres et a la particularité de supporter des communications non bloquantes. La vérification du RoC se fait par simulation sur ordinateur et par une implémentation sur FPGA. Dans l'implémentation, le RoC est connecté avec plusieurs ressources qui permettent de générer des communications au réseau. Ces ressources sont constituées de processeurs, de mémoires, de contrôleurs, etc. Tel que mentionné précédemment, le RoC est configurable. Cet aspect configurable a d'ailleurs facilité l'interconnexion du RoC avec un système de communication interpuce, le tunnel HyperTransport. Un ensemble d'analyses sont effectuées sur le RoC ainsi que sur sa plateforme pour une implémentation sur FPGA.

Nous présentons les résultats de synthèse selon différents paramètres de configuration et démontrons que l'augmentation de l'espace occupé par le design est moins que quadratique selon le nombre de nœuds tout offrant ayant une augmentation quasi linéaire de la bande passante globale. Pour 8 nœuds ou moins, la complexité du RoC est moindre que celle d'une topologie en maille. Pour 12 nœuds et plus, le RoC consomme plus de ressources matérielles du FPGA que la maille. Par contre, il ne faut pas oublier que le RoC supporte simultanément plusieurs communications entre une paire de source et destination unique, ce qui n'est pas le cas pour la maille. Il est à noter que la latence du RoC est 3,7 fois moindre que la maille. L'implémentation du RoC s'approche des performances d'une topologie en connexion point à point, mais en utilisant moins d'espace. L'utilisation d'espace est inférieure à 25% des ressources sur une puce FPGA VP100 de Xilinx pour un RoC à 16 nœuds quesupporte une bande passante agglomérée de 6 Go/s. Finalement, nous présentons différents résultats de simulations effectuées pour l'intégration du RoC avec le tunnel HyperTransport.

ABSTRACT

As the microelectronic industry evolve, the need for more efficient communication networks is important to the success of new System on Chip (SoC) designs. These systems are complex. To manage that complexity, they must be divided into multiple blocks: processors, memories, I/O controllers, specialized hardware blocks, etc. We present an FPGA prototype implementation of a Rotator-on-Chip (RoC), a simple and scalable novel network-on-chip based on the token-ring concept developed in collaboration with STMicroelectronics in Ottawa. The reported prototype design is generic with respect to the number of nodes, data width and address space. Nowadays, several architectures and some commercial products exist, but NoC implementation results are scarce. Our architecture differs from others in the literature, but still uses important basic concepts of NoCs. We first summarize these concepts and review related works. The RoC functionality and the details of our FPGA validation platform are presented. Then we report synthesis results showing a less-than-quadratic area growth with respect to the number of nodes, yet with a quasi-linear aggregate bandwidth growth. For 8 nodes or less, hardware complexity of the RoC is lower than the mesh implementation. For 12 nodes and beyond, the RoC consumes more FPGA resources than a mesh. However, the RoC can support simultaneously all possible one-to-one connections between source and destination nodes, which is not the case for the mesh. In terms of latency, the RoC latency is shown to be 3.7 times lower than that of a mesh. The RoC implementation approximates the performance of a crossbar but uses much less area. The slice utilization is less than 25% of those available on a Xilinx VP100 for a 16 node version of the RoC, supporting an aggregate bandwidth of about 6 GB/s. The implementation has been validated by simulations and implemented on a FPGA development board.

TABLE DES MATIÈRES

LISTES DES FIGURES

LISTE DES TABLEAUX

LISTE DES ACRONYMES

AMBA	: Advanced Microcontroller Bus Architecture
ASIC	: Application Specific Integrated Circuits
AXI	: Advanced eXtensible Interface
BRAM	: Block Random Access Memory
CRC	: Cyclic Redundancy Check
DDR	: Double-Data-Rate
DSP	: Digital Signal Processor
EDK	: Embedded Development Kit
FFD	: Flip-Flop D
FIFO	: First In First Out
FPGA	: Field-Programmable Gate Array
Gb/s	: Gigabit par seconde
Go/s	: Gigaoctet par seconde
HT	: HyperTransport
HyRoC	: Hyper Ring on Chip
IP	: Intellectual Property
I/O	: Input / Output
Mb/s	: Mégabit par seconde
Mo/s	: Mégaoctet par seconde
MHz	: Mégahertz
MPSoC	: Multi-Processor System on Chip
NoC	: Network on Chip
OCP	: Open Core Protocol
OPB	: On-chip Peripheral Bus
PC	: Personal Computer
QoS	: Quality of Service

RoC	: Rotator on Chip
SDRAM	: Synchronous Dynamic Random Access Memory
SoC	: System on Chip
SPIN	: Scalable Programmable Interconnection Network
UART	: Universal Asynchronous Receiver Transmitter
UCF	: User Constraint File
VHDL	: VHSIC Hardware Description Language
µm	: micromètre

LISTE DES ANNEXES

INTRODUCTION

Au fur et à mesure que le nombre de transistors augmente dans une puce électronique, il sera difficile pour un concepteur d'utiliser pleinement les nouvelles ressources matérielles pour la conception d'un SoC (*System on Chip*). Dans le but de réduire la complexité de ces systèmes et ainsi d'en faciliter la conception, il est nécessaire d'identifier les aspects qui composent leur complexité. Principalement deux catégories regroupent ces aspects. La première est liée à la complexité des technologies d'intégration utilisées et la seconde comprend plusieurs aspects qui influencent la méthode de conception de ces systèmes. Diviser le système en plusieurs blocs est une des méthodes fréquemment utilisées pour gérer cette complexité. De tels blocs peuvent être des processeurs, des mémoires, des contrôleurs d'entrées et de sorties et des blocs matériels spécialisés. Pour que l'ensemble du système fonctionne, il est nécessaire que ces blocs soient connectés entre eux. C'est par l'entremise d'un réseau de communication que les blocs s'échangeront les données.

Le SoC est composé de plusieurs blocs qui sont disposés de façon optimale les uns par rapport aux autres. Pour connecter les blocs entre eux, il existe différentes architectures pour établir le lien de communication. Afin de répondre aux multiples besoins des SoC, l'architecture doit prendre en compte divers critères comme le coût, la performance, la flexibilité, l'extensibilité, etc. Le bus et la connexion point à point sont des exemples d'architectures fréquemment utilisées comme lien de communication dans la conception des systèmes.

Plusieurs variantes d'architectures en bus existent. Elles sont simples et fonctionnent toutes sur une méthode de demande de permission pour communiquer. Par contre, leur piètre capacité d'extensibilité et leur grande contention qui provient des communications bloquantes peuvent subvenir dans un bus et limitent leur utilisation dans la conception de

SoC. Par contre, la connexion point à point est celle qui offre les plus petits délais en communication. La communication grâce à des connexions point à point est non bloquante et supporte simultanément plusieurs communications. D'un autre côté, l'extensibilité de cette architecture est limitée et son coût matériel est élevé. Pour conclure, le bus et la connexion point à point ne sont pas parfaitement adaptés pour répondre aux besoins des SoC.

Pour ce faire, un nouveau paradigme est apparu pour répondre aux besoins des SoCs, les réseaux intégrés sur puce (en anglais, *Network on Chip*, NoC). Ces réseaux sont proposés pour remplacer les bus et les connexions point à point comme méthode de communication dans les SoC. Ils permettent de limiter l'utilisation de l'espace physique et d'éviter les problèmes de synchronisation d'horloge entre différentes composantes. Leur transmission de donnée se fait par paquets d'une source à sa destination. Les paquets sont routés dans le réseau selon un algorithme propre à chaque NoC. Ils peuvent être fragmentés en plusieurs cellules ou conserver une taille fixe. L'échange des données entre les composantes et le réseau nécessite une enveloppe d'interfaçage qui permet l'utilisation de différents protocoles de communication.

De nos jours, plusieurs architectures et produits commerciaux existent, mais peu de détails sur ces réseaux et leur implémentation matérielle sont révélés. Dans le but de faire la lumière sur le domaine des réseaux intégrés sur puce, ce mémoire passe en revue différentes architectures et présente notre réseau intégré sur puce RoC (*Rotator on Chip*) développé en partenariat avec STMicroelectronics de Ottawa. Conçu avec le langage de description de matériel VHDL (*VHSIC Hardware Description Language*), ce réseau est simple, extensible et est basé sur un modèle utilisant une topologie en anneau. Le RoC est configurable selon différents paramètres et il a la particularité de supporter des communications non bloquantes. Il peut ainsi gérer simultanément toutes les

communications du réseau si celles-ci sont établies entre une paire de source et de destination unique.

Le RoC a d'abord été conçu à un haut niveau d'abstraction avec le langage SystemC pour faciliter sa réalisation. Par la suite, dans le but d'obtenir des données tangibles sur les capacités du réseau, un modèle a été réalisé au niveau matériel. Ce modèle considéré de bas niveau, a été conçu pour une implémentation sur un FGPA (*Field-Programmable Gate Array*) de type Xilinx. Trois modèles de bas niveau existent pour le Roc et un aspect intéressant de sa réalisation a été de le comparer à d'autres architectures NoC. Le RoC est configurable selon différents paramètres. Il l'est pour la largeur de ses paquets, pour son système d'adressage et pour son nombre de connexion aux composantes du système. Ses paramètres sont configurables et ils permettent d'ajuster son architecture. De cette façon, il est possible d'obtenir des implémentations différentes pour comparer le RoC avec d'autres architectures.

La conception du RoC et des ressources de sa plateforme pour le tester ont été réalisées avec les outils ISE et EDK (*Embedded Development Kit*) de la compagnie Xilinx [XILI00]. Les ressources du système consistent en plusieurs processeurs et périphériques. Les périphériques sont les mémoires, les contrôleurs, etc. L'ensemble des composantes d'une ressource permet de générer les communications pour tester le réseau. Pour établir une communication avec le réseau, les ressources doivent être interfacées selon un protocole de communication standard. Cette méthode permet de réutiliser le code et simplifier les communications avec d'éventuelles autres composantes électroniques. Pour ce faire, une enveloppe a été développée pour que le RoC isole la complexité du protocole des communications. L'enveloppe gère les communications et achemine ainsi les requêtes des ressources vers les ports d'entrées et de sorties du réseau.

Tel que mentionné précédemment, le RoC est configurable selon plusieurs paramètres et permet d'ajuster le RoC selon le nombre de ressources connectées au réseau. Cet aspect configurable a d'ailleurs facilité l'interconnexion du RoC avec un système de communication interpuce, le tunnel HT (*HyperTransport*). Un pont traduit les communications entre le tunnel et les deux réseaux RoCs. Il se connecte aux FIFOs (*First In First Out*) d'entrée et de sortie des réseaux et transfère les paquets de façon transparente. Ainsi, les ressources appartenant à un RoC différent s'échangent des données même si elles ne sont pas sur un réseau identique.

Après la conception du RoC et de son enveloppe, il est nécessaire d'effectuer des simulations sur ordinateur avec l'outil Modelsim [MENT00] pour vérifier le fonctionnement du réseau. Le simulateur permet essentiellement de déceler les erreurs de conception. Deux étapes de vérification doivent être franchies avant de pouvoir passer à l'implémentation FPGA. Tout d'abord, il faut que la simulation seule du réseau puisse fonctionner avec une génération simple de paquets à travers les nœuds. Cette génération de paquet se fait par l'entremise d'un banc de test en VHDL. Les erreurs décelées à cette étape doivent être résolues avant de passer aux simulations des ressources qui seront connectées aux RoC. La seconde étape consiste à générer de nombreux paquets qui transitent à travers le réseau. D'autres erreurs peuvent être décelées dans la conception et sur le synchronisme des communications. Ces situations ne sont pas toujours remarquées à l'étape précédente. Après chacune des étapes de vérification, il est important d'effectuer des simulations pour chacune des étapes de synthèse. Une simulation comportementale ne prend pas en compte les délais que peut avoir une communication dans un modèle post-synthèse du réseau. La vérification complète du réseau étant terminée, il est possible de passer à l'implémentation du réseau sur le FPGA. Encore là, des tests doivent être effectués pour s'assurer du fonctionnement du réseau dans une implémentation matérielle. Même si la simulation après placement et routage est réussie, certains comportements physiques dans le FPGA peuvent influencer le fonctionnement du réseau.

Contributions de ce mémoire

Le modèle haut niveau en SystemC développé en collaboration avec STMicroelectronics de Ottawa a seulement servi pour une exploration architecturale. Le but premier des travaux effectués dans ce mémoire était d'obtenir des informations réelles sur les performances du RoC. Ces informations ont permis de comparer notre architecture face à d'autres NoC. D'ailleurs, peu de NoCs sont détaillés en profondeur dans la littérature. Ce mémoire permet d'éclairer tout nouveau concepteur au domaine des NoCs. La conception à bas niveau a permis d'amener une nouvelle vision au développement de notre modèle et d'ainsi confirmer qu'il est possible de réaliser avec un faible coût matériel un NoC avec une architecture non bloquante. Le fait que notre architecture est inspirée du modèle des registres à décalage est un élément nouveau. Ce n'est pas l'architecture qui était jugée favorite au début de la conception du RoC matériel, mais elle s'est avérée plus qu'intéressante. La conception à bas niveau a permis d'apporter de nouvelles idées. La fonctionnalité des matrices de mémoires et le mécanisme de transfert des paquets entre le nœud et la banque ne sont pas identiques au modèle de haut niveau. Au lieu de transmettre un seul paquet par pivot, le RoC implémenté permet de transmettre un maximum de paquets aux banques à chaque pivot. Cela réduit théoriquement la congestion et la latence dans le réseau. En plus de fournir des informations sur les différents aspects du RoC à STMicroelectronics, cette implémentation a permis d'apporter une expertise dans notre groupe de recherche pour le design sur FPGA avec les outils ISE et EDK. Le RoC est présentement réutilisé pour des recherches plus approfondies par un étudiant au postdoctorat. Un autre aspect important des travaux est celui qui a permis l'interopérabilité avec le système de communication interpuce HyperTransport. La partition d'une application entre plusieurs puces n'est pas un aspect nouveau [DESL05], mais la méthode utilisée pour faire communiquer ces systèmes distribués sur plusieurs puces l'est. Des systèmes peuvent souvent inclure des liens de communication interpuce, mais la solution utilisée est nouvelle du fait qu'elle intègre de façon transparente la communication interpuce comme si elle était intra puce. Pour ces

raisons, le travail effectué fait présentement l'objet d'une rédaction pour un article de revue par Ami Castonguay [CAST06].

Plan du mémoire

Une revue de littérature fait le tour du sujet des réseaux intégrés sur puce. Les besoins qui justifient l'utilisation des NoCs sont introduits avec quelques explications de certains aspects des SoC. Pour comparer les différentes architectures de communication, un compte rendu de plusieurs NoC implémentés est présenté. Le chapitre 2 est la base de notre réseau intégré sur puce. L'architecture du RoC est différente de celle des NoC trouvés dans la littérature, mais elle utilise tout de même plusieurs concepts de base des NoC. Une vue d'ensemble de son mécanisme de communication est expliquée par l'entremise de ses concepts de nœuds et de banques. Différents modèles du RoC sont présentés avec des fonctionnalités et des méthodes d'optimisations. La présentation du RoC se poursuit au chapitre 3 par la présentation de l'architecture matérielle. Celle-ci est utilisée pour l'implémentation sur FPGA. Le canal de communication, l'enveloppe de communication et les paramètres configurables du réseau sont des concepts importants vus dans ce chapitre. Le chapitre 4 concerne l'aspect implémentation sur FPGA et la méthode pour tester le RoC. La plateforme du système multiprocesseurs qui sert à tester le fonctionnement du réseau est présentée tout en expliquant la méthode pour la concevoir. Elle est suivie d'une courte explication sur le fonctionnement de l'application exécutée sur la plateforme. En dernier lieu, la méthode avancée pour intégrer le RoC avec le moyen de communication interpuce HyperTransport est présentée et suivie des spécifications du pont qui permet l'interconnexion. Le chapitre 5 décrit l'ensemble des analyses faites sur le RoC ainsi que celles relatives à l'implémentation de la plateforme sur FPGA. Nous présentons les résultats de synthèse selon les différents paramètres de configuration. Nous obtenons des informations sur l'espace matériel occupé, la bande passante, la latence des communications, etc. Finalement, nous présentons différents résultats des simulations effectuées pour l'intégration du RoC avec le tunnel HT.

CHAPITRE 1

REVUE DE LITTÉRATURE

1.1 Évolution de la microélectronique

Depuis plus de 30 ans, la microélectronique n'a cessé d'évoluer et de repousser les limites des systèmes électroniques. Grâce aux avenues technologiques des semi-conducteurs, il est de nos jours possible d'intégrer plusieurs systèmes dans une seule puce électronique. Cette solution fait appel à des architectures nommées système intégré sur puce (SoC) qui consistent à interconnecter avec un très haut débit de communication plusieurs composants hétérogènes. L'intégration des composants améliore le rendement des systèmes et permet la création de nouvelles applications pour les produits commerciaux.

Réaliser l'intégration de touts ces composants pour qu'ils fonctionnent en symbiose est un défi sur plusieurs plans. La technologie des semi-conducteurs s'améliore, les transistors rapetissent, les outils de conception et les architectures évoluent. Tous ces facteurs influencent la conception et rendent la réalisation du système de plus en plus complexe.

Cette complexité nécessite un temps important de conception que les techniques actuelles ne sont pas parfaitement adaptées à prendre en charge. Jumelée au nombre croissant de nouveaux transistors disponibles dans une puce électronique à chaque année, cette différence crée un écart entre ce qu'offre la technologie et ce que le concepteur est capable d'utiliser. Ce phénomène est ce que nous appelons l'écart de productivité. Au paravant, le temps de conception d'un système suivait l'évolution des semi-conducteurs et permettait d'utiliser tous les transistors d'un circuit. Pour rattraper l'écart de

productivité qui se présente, il est indispensable d'améliorer le taux de productivité. Le taux de productivité est le rapport entre la conception réalisée et les moyens mis en œuvre pour y parvenir (e.g. nombre de transistors/homme-année). Pour cela, de meilleurs outils doivent être conçus et notre façon de concevoir les systèmes doit être repensée. Une autre façon d'améliorer le taux de productivité est de diviser la complexité et de réutiliser les composants (bloc, éléments de calcul, etc.). Ces composants peuvent être répliqués plusieurs fois dans un seul design ou utilisées dans d'autres versions de design pour sauver du temps de conception.

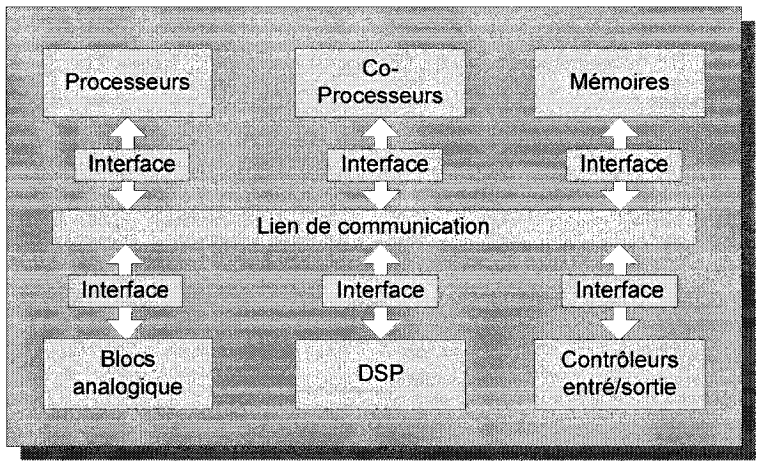


Figure 1-1. Modèle d'un système sur puce avec ses composants

Tel qu'illustré à la **Figure 1-1**, les composantes d'un système peuvent être de type processeur, coprocesseur, processeur de traitement de signaux (en anglais, *Digital Signal Processor*, DSP), bloc de mémoire, compteur, interface de communication externe, bloc matériel dédié et bloc analogique. Pour faire communiquer entre eux cet amalgame de composantes, un élément d'interconnexion doit relier ces dernières. Plusieurs solutions s'offrent pour résoudre ce besoin de communication.

1.1.1 Problématiques des SoC et architectures désuètes

Les applications exécutées sur SoC nécessitent de plus en plus de performance. Tel que mentionné précédemment, pour réaliser de tels systèmes, nous devrons revoir la façon de concevoir les puces électroniques. Leur complexité relève de plusieurs facteurs et les identifier en facilitera la conception. Ces facteurs peuvent être regroupés en deux catégories principales. La première est liée à la complexité des semi-conducteurs utilisés et la seconde comprend divers aspects influençant la méthode de conception des systèmes.

Complexité des semi-conducteurs : Le changement vers une technologie de semi-conducteur plus avancé permet de miniaturiser la puce électronique et d'améliorer de façon importante ses performances. Par contre, ce changement de technologie apporte des complexités additionnelles. Le comportement physique de la puce doit être analysé et parfaitement maîtrisé, car certains comportements nouveaux peuvent subvenir lorsque les transistors sont rapetissés. De plus, il se peut que la technologie nécessite l'utilisation de nouveaux matériaux. Ces matériaux peuvent être utilisés par exemple pour limiter les fuites de courant.

Complexité des systèmes : La réalisation de ces SoCs nécessite de nouvelles méthodes de conception. L'introduction de composants réutilisables, l'utilisation de nouvelles architectures et l'utilisation de plusieurs logiciels embarqués fonctionnant sur des processeurs différents ajoutent de la complexité au flot de conception. Les outils doivent supporter ces méthodes tout en restant simples aux yeux du concepteur. Dans le meilleur des cas, les outils offrent des choix au concepteur ce qui lui permet d'ajuster le flot de conception selon les besoins du design. Par exemple, optimisation en fonction des coûts, en fonction d'une fréquence d'opération désirée, etc. Cinq autres facteurs qui font partie de la complexité des systèmes résumée sont survolés.

Plateforme de développement : La venue de nouveaux systèmes nécessite l'utilisation de nouvelles plateformes de développement. Ces plateformes permettent de tester de plus gros systèmes et permettent d'y faire de nouvelles connexions qui n'étaient pas disponibles auparavant (Ethernet, vidéo, mémoires plus grandes).

Productivité de conception : Elle est un des facteurs majeurs à améliorer pour réussir à suivre l'évolution des semi-conducteurs. Il est inévitable d'élever le niveau d'abstraction de la conception pour simplifier le système. Les outils devront s'adapter et fournir davantage d'options pour l'optimisation.

Source d'alimentation : Certains produits ont des requis différents ayant des répercussions sur la consommation de puissance: design pour faible consommation, communications sans fil, applications multimédias, etc.

Test et vérification : La complexité et la diversité des composantes étant énormes, il faut développer de nouvelles méthodologies de tests. Chaque bloc IP (*Intellectual Property*) doit être fourni avec son propre banc de test pour faciliter le déverminage du SoC. Ce point est très important, car il ne faut pas oublier que de nos jours 70% du temps de conception est investi dans le test et la vérification.

Plateforme de développement et réutilisation d'IPs : La venue de nouveaux systèmes nécessite l'utilisation de nouvelle plateforme de développement. Ces plateformes permettent de tester de plus gros systèmes et permettent d'y connecter de nouvelles connexions qui n'étaient pas disponibles auparavant (Ethernet, vidéo, mémoires plus grandes). Utiliser des IPs différents peut s'avérer difficile et consommer un temps considérable dans toutes les étapes du flot de conception. Ces IPs peuvent avoir été

conçus selon différentes technologies et divers niveaux d'abstraction. Une solution est d'utiliser des enveloppes (*wrappers*) pour simplifier l'intégration selon différents protocoles de communications. Établir une synergie entre toutes ces composantes sera nécessaire et requerra un médium de communication pour l'échange des données.

Un SoC est composé de plusieurs composantes disposées de façon optimale l'une par rapport aux autres. L'architecture du SoC doit ainsi être soigneusement étudiée. Pour connecter les composantes entre elles, il existe différentes architectures qui permettent de faire le lien de communication. Ce lien doit répondre aux multiples besoins du système à concevoir. La meilleure architecture doit prendre en compte des critères comme le coût, la performance, la flexibilité, l'extensibilité, etc. Ces critères seront étudiés à la section 1.2 lors de notre présentation des architectures de communication. Mais auparavant, commençons par les deux principales architectures utilisées dans les systèmes conventionnels.

1.1.1.1 Le bus

Une première approche d'architecture pour répondre au besoin en communication est d'utiliser une architecture de bus. La **Figure 1-2** est un exemple de communication typique. Encore utilisé de nos jours dans la plupart des ordinateurs et systèmes électroniques, le bus a les particularités d'avoir une latence faible (petits délais de communication) et d'interconnecter de façon simple les composantes du système. Divers standards de bus existent et ils sont tous basés sur une méthode de demande de permission pour communiquer (arbitrage). Une composante qui demande une permission de communiquer à l'arbitre est nommée un *maître*. De l'autre côté, lorsque le maître obtient la permission, il communique avec un *esclave* du bus. Cet esclave est tout simplement une autre composante faisant partie du système. Par exemple, dans un système utilisant une communication en bus, le processeur est considéré comme un maître et la mémoire comme un esclave. Selon les caractéristiques du système, il est

nécessaire que le bus soit constitué de niveaux de communication différents (e.g. un bus par niveau). Tel que détaillé à la **Figure 1-2**, un bus de *haute performance* permet d'offrir un débit de communication important pour répondre au besoin de son processeur. Si plus d'un processeur, ou microcontrôleurs sont nécessaires dans le système, alors ces deniers sont connectés à l'ensemble du système par un deuxième bus (niveau), nommé bus des *périphériques*.

Différents produits commerciaux offrent le bus comme solution IP pour les SoC. Il existe le bus AMBA (*Advanced Microcontroller Bus Architecture*) de ARM [ARM01] et le bus CoreConnect™ d'IBM [IBM03]. La piètre capacité d'extensibilité et la grande contention provenant des communications bloquantes qui peuvent subvenir dans un bus limitent son utilisation dans la conception de SoC. La communication bloquante survient dans le bus lorsqu'une communication vers un esclave empêche que tout autre maître puisse utiliser le bus comme moyen de communication. Cette situation est partiellement contournable dans l'exemple des communications séparé (en anglais, *split transaction*) pour le bus AMBA. Même si le bus est partagé parmi tous les maîtres et qu'un seul de ceux-ci peut communiquer à la fois, la communication séparé permet à un maître de s'intercaler dans une communication en obtenant temporairement les droits sur le bus. Il est fort probable qu'avec les années, un SoC incorpora des dizaines et même des centaines d'éléments de calculs. On entend par élément de calcul, toutes composantes matérielles, processeurs ou autres qui effectuent des calculs du système. Un SoC ne pourra fonctionner avec le partage d'une bande passante pour un seul bus ou même une hiérarchie de bus.

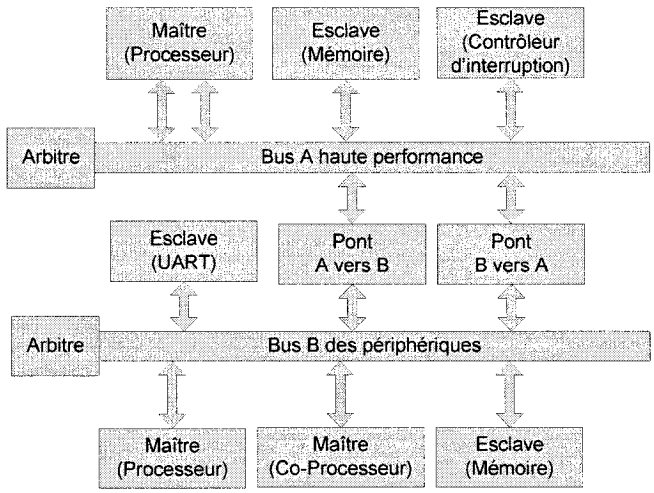


Figure 1-2. Modèle du bus avec des composantes

1.1.1.2 Le crossbar

Un second modèle de communication pour les SoCs est celui des connexions point à point (en anglais, *crossbar*), **Figure 1-3**. Cette architecture est celle qui offre les plus petits délais de communication. Peu de temps est perdu dans la transmission des données, car les connexions sont directes entre chacune des composantes. Un réseau comme le *crossbar* pouvant supporter toutes les connexions sans que celles-ci ne se bloquent est appelé un réseau non bloquant. Ce réseau peut supporter simultanément toutes communications d'une composante ou plus sans générer de conflits dans les liens de communications [FENG81]. Le crossbar est dispendieux lorsque l'on calcule son coût matériel pour chaque bit transmis. Également, ses transferts de données dans les designs actuels peuvent occasionner des problèmes. En effet, il a été démontré que parfois, la propagation des signaux est si grande qu'elle peut s'étendre sur une période complète d'horloge [BEDE02]. Le crossbar de base fonctionne avec une seule fréquence d'opération. Si les chemins entre ses composantes sont trop grands, sa fréquence maximale est réduite et influence la performance générale du système. Il faut mentionner aussi que l'extensibilité du crossbar est limitée. Son architecture a une complexité matérielle de $O(N^2)$ où N est le nombre de composantes. Finalement, le format des

données n'est pas adaptable pour différentes composantes. Cela rend l'ajout de composantes coûteux et limite la réutilisation d'IP.

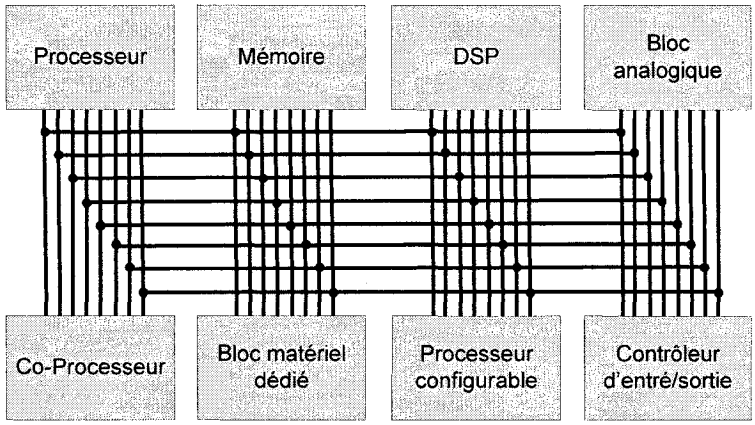


Figure 1-3. Modèle du Crossbar avec ses composantes

1.2 Paradigme des réseaux sur puce

Un nouveau paradigme, les réseaux intégrés sur puce (NoC), a été proposé pour remplacer les bus et les méthodes d'interconnexion dédiée [BEDE02b]. Les NoCs sont des réseaux de communications permettant une connexion physique entre plusieurs blocs dans un environnement SoC. Ils permettent de limiter l'utilisation de l'espace physique et d'éviter les problèmes de synchronisation d'horloge entre différentes composantes. La transmission des données se fait par paquets de la source à la destination. Ils sont routés dans le réseau selon un algorithme propre à chaque NoC. Les paquets peuvent être fragmentés en plusieurs cellules (en anglais, *flit*) ou conserver une taille fixe. Pour échanger les données avec les composantes, celles-ci sont connectées au NoC et requièrent une enveloppe d'interfaçage pour permettre l'utilisation de différents protocoles de communication [PDGI05].

La structure d'un NoC est une série de commutateurs ou routeurs interconnectée par des canaux de communication. La manière dont ces commutateurs sont reliés définit la

topologie du réseau, alors que le moment et la direction vers laquelle un paquet doit prendre un chemin déterminent le fonctionnement du commutateur. La topologie et le fonctionnement des commutateurs ont des impacts directs sur les capacités de performance du NoC, latence et bande passante agglomérée.

Même si les NoC apparaissent comme étant la solution pour remplacer les bus pour la communication dans les SoC, ils devront prouver qu'ils sont économiquement viables [ARTE06]. Plus précisément, pour être viable ils devront :

- Réduire le coût de production du SoC
- Augmenter la performance du SoC
- Réduire le temps pour atteindre le marché
- Écourter le temps avant la production de masse
- Réduire les risques de conception du SoC

1.2.1 Topologies existantes

D'une certaine façon, la topologie des NoC ressemble à d'autres réseaux existant sous diverses formes. Pensons au réseau téléphonique, les ordinateurs à l'intérieur d'une entreprise, le système électrique, l'Internet, etc. Certains disposent d'une structure régulière et d'autres sont plutôt adaptés aux configurations du milieu. Plusieurs topologies existent, mais elles ne sont pas tout à fait adaptées à la problématique des SoC. Il serait donc intéressant de revoir ces modèles qui ont été développés dans les dernières décennies et les appliquer pour bâtir de nouvelles architectures réseaux pour les communications intra puce. Parmi les topologies NoC les plus populaires, nous retrouvons celle des bus, crossbar, papillon, arbre élargi, anneau et maille. Le bus et le crossbar (et leur limitation) ont déjà été présentés (section 1.1.1.1 et 1.1.1.2). Dans ce qui suit, un résumé des autres topologies et de leurs caractéristiques est présenté. Dans les

prochaines figures, R désigne un routeur et N désigne un nœud. Un routeur est considéré comme un commutateur et les nœuds peuvent être une ou des composantes (processeur, mémoire, etc.)

1.2.1.1 Papillon

Peu de topologies papillon sont présentent dans la littérature en raison du grand nombre d'interconnexions et de leur difficulté d'extensibilité. Leur diamètre étant logarithmique en fonction du nombre de composantes connectées, cette topologie entraîne une complexité de O(N log N). Sa structure est telle qu'il n'existe qu'un seul chemin pour chaque paire de source et destination. La latence moyenne est O(log N). La **Figure 1-4** présente la topologie papillon et sa variante. La variante de la topologie est celle du réseau de Benes qui utilise deux réseaux papillon collés dos à dos pour diminuer les probabilités que deux transactions entrent en conflit à un commutateur.

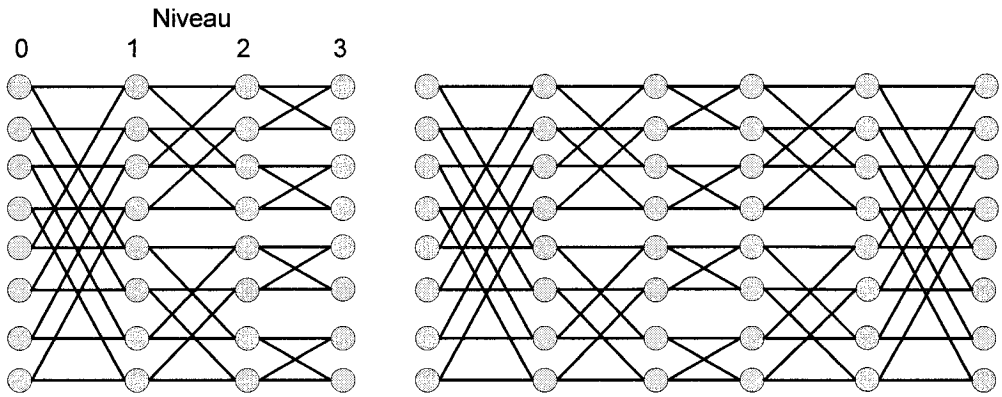


Figure 1-4. a) Modèle de la topologie Papillon b) Modèle du réseau de Benes

1.2.1.2 Arbre élargi

Ayant lui aussi un diamètre logarithmique pour une complexité de O(N log N) et une latence faible O(log N)), l'arbre élargi ressemble à une topologie papillon ayant des liens bidirectionnels. L'arbre est structuré de différents niveaux de commutation. La **Figure**

1-5 présente la topologie classique de l'arbre élargi avec une seconde présentation selon un arbre élargi à deux niveaux. Normalement, quatre composantes se connectent à un premier commutateur. Ensuite, chaque commutateur se connecte à un niveau supérieur de commutation qui comporte lui aussi le même ratio de composantes par commutateur. Le nombre de composantes par commutateur est fixe et détermine le nombre de niveaux. L'arbre élargi nécessite moins de commutateurs que le papillon. SPIN (*Scalable Programmable Interconnection Network*) [GUGR00] est un exemple de réseau basé sur la topologie en arbre. Le routage de l'arbre est simple au détriment que ses communications sont bloquantes, ce qui limite sa bande passante. De plus, son coût est élevé. Comme on peut le remarquer, le modèle classique prend quasiment autant de routeurs que de nœuds.

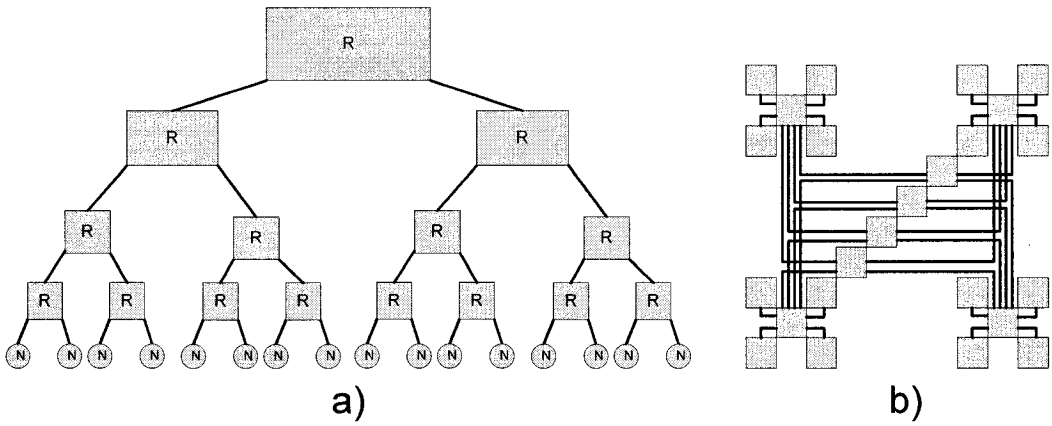


Figure 1-5. a) Modèle classique de la topologie en arbre b) Modèle de l'arbre élargi à deux niveaux

1.2.1.3 Anneau

Représentée par une boucle fermée, chaque composante connectée à l'anneau a deux voisins contigus et échange les paquets avec ses deux voisins dans un sens horaire ou antihoraire (**Figure 1-6**). Le chemin des paquets étant fixe, un seul chemin ne peut être emprunté. La complexité et la latence du réseau sont O(N). Il est possible d'utiliser des

liens de communication bidirectionnelle pour diminuer considérablement la latence. L'architecture en anneau étant extensible, il requiert simplement la modification du lien entre deux composantes. La méthode de routage n'a pas à être modifiée. La configuration de l'anneau ne permet pas les communications bloquantes. Cela évite ainsi les goulots d'étranglement.

Proteo [SINU02] est un exemple de topologie en anneau. La topologie de *Chordal* est une version modifiée de l'anneau et Octagon [KNDR01] l'utilise. Généralement, la structure *Chordal* établit des liens entre un commutateur et ses commutateurs opposés pour diminuer la latence (**Figure 1-6b**). L'anneau est une topologie davantage utilisée pour les architectures réseau des années 80. De nos jours, il existe différentes puces SoC incorporant cette topologie (voir section 1.3.1).

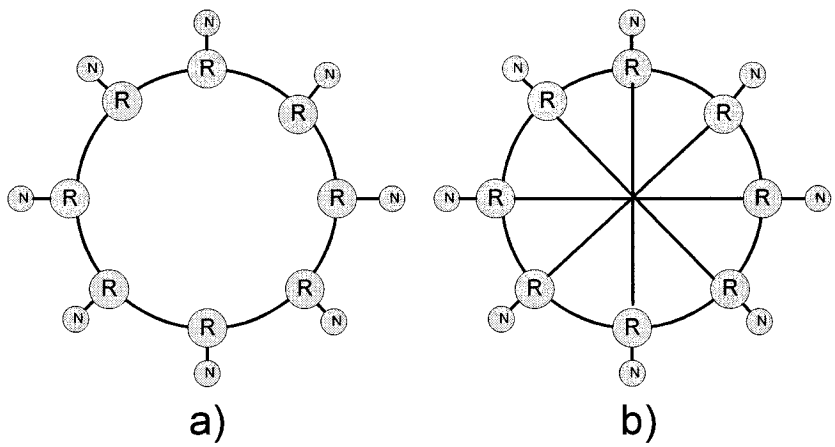


Figure 1-6. a) Modèle classique d'une topologie en anneau b) Modèle en anneau de la structure de chordal

1.2.1.4 Maille

Sa topologie est basée sur le concept du routage de paquets et elle incorpore un algorithme de routage pour chacun de ses commutateurs (en anglais, *switches*).

Structurellement, la maille ou maille 2D (en anglais, 2D *mesh*) ressemble à un grillage. Présentés à la **Figure 1-7a**, des commutateurs se trouvent aux intersections et sont connectés à des composantes, souvent appelées nœuds (en anglais, *nodes*). Le routage des paquets se fait au hasard ou selon un algorithme pour déterminer les chemins les moins congestionnés. D'autres architectures identifient leurs composantes par donnée cartésienne, une coordonnée en X (ligne) et une en Y (colonne). Ces mailles utilisent les coordonnés pour leur algorithme de routage appelé XY. Cet algorithme route les paquets avec un déplacement en X et lorsque le paquet est dans la bonne colonne, le paquet monte ou descend selon sa coordonnée Y. Contrairement aux dernières topologies mentionnées, un paquet envoyé sur un réseau en maille a la possibilité d'emprunter n'importe quel chemin de la source à la destination. La complexité d'un réseau en maille est O(N) et sa latence est $O(\sqrt{N})$.

De nombreuses architectures NoC sont basées sur les mailles. Leur coût de connexion est raisonnable et la régularité de leur placement se porte bien à la technologie ASIC (*Application Specific Integrated Circuits*). De plus, la maille offre un grand débit de bande passante. Cliché [KJSF02] et Aethereal [RGRM03] sont des exemples d'architectures basées sur les mailles. Des variantes à cette topologie sont la structure en tore de Dally [DATO01] et Marescaux [MBVV02] (**Figure 1-7b**).

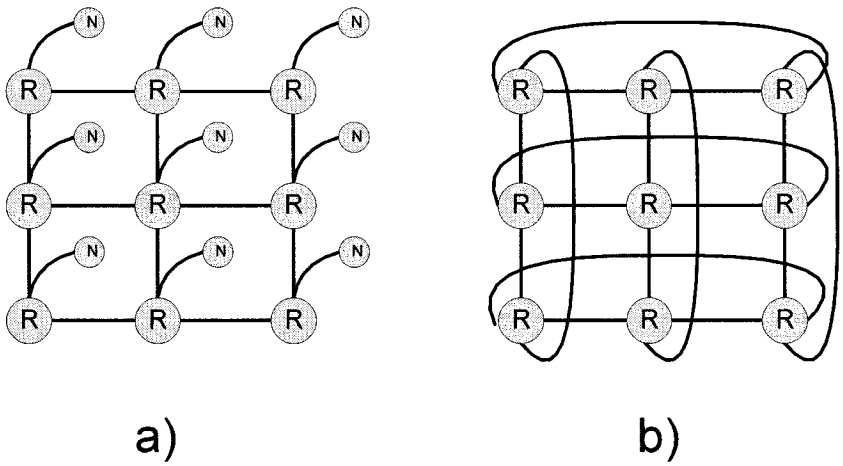


Figure 1-7. a) Topologie en maille avec ses composantes b) Topologie en tore de la maille

1.2.2 Routage

Le chemin qu'empruntera un paquet de sa source à sa destination est déterminé par l'algorithme de routage. Les algorithmes peuvent changer en fonction des applications exécutées sur un réseau. Les commutateurs exécutent l'algorithme de routage et chaque NoC fonctionne différemment. Principalement trois sortes d'algorithmes de routage existent [YOO03].

1.2.2.1 Stockage et réémission (store and forward)

C'est le plus simple des algorithmes à implémenter. Il utilise beaucoup d'espace mémoire et sa méthode de routage ralentit les communications, car il nécessite la confirmation des transmissions. Pour qu'un paquet passe d'un commutateur à l'autre, le premier commutateur dans la chaîne doit recevoir complètement le paquet et par la suite demander au second commutateur s'il est prêt à recevoir le paquet. Si ce n'est pas le cas, le commutateur devra garder en mémoire le paquet et redemander la permission de le transmettre.

1.2.2.2 Par raccourcis (virtual-cut-through)

Semblable au précédent algorithme, il nécessite autant de mémoire. Par contre, la latence est réduite, car le transfert d'un commutateur à un autre peut débuter avant même que le paquet ne soit totalement arrivé au premier commutateur.

1.2.2.3 Trou de ver (wormhole)

Il est l'un des meilleurs algorithmes de routage pour ses résultats de latences réduites. Aussitôt que les informations de routage d'un paquet sont disponibles (souvent placé à la tête d'un paquet), le prochain commutateur sur le chemin pourra commencer à mémoriser ce paquet et chercher à établir une nouvelle commutation. Un commutateur acceptant une transmission à son entrée ne pourra libérer la connexion qu'il établira entre son port d'entrée et son port de sortie que lorsqu'il aura complètement transféré son paquet à son port de sortie. Cela implique que son port d'entrée sera indisponible. Il est à noter qu'un tel algorithme de routage est plus susceptible aux communications bloquantes, car des paquets devant prendre le chemin d'un commutateur bloqué se verront refuser à répétition la permission de commutation.

1.3 Architectures implémentées de NoC

Nombreux sont les articles qui discutent des avantages pratiques des NoCs et qui présentent leur modèle avec des détails de fonctionnement. Il est par contre difficile de trouver des implémentations matérielles de ces modèles. Peu de résultats et de détails techniques des implémentations sont fournis.

Le Tableau 1-1 rassemble les différents réseaux afin de fournir une image d'ensemble de l'état de l'art des implémentations FPGA et ASIC de NoC. Il est à noter que les informations ne sont pas exhaustives et que seuls les NoCs les plus connus sont passé en revue. Chaque rangée représente les caractéristiques d'un NoC. Les colonnes topologie,

routage et configuration détaillent l'architecture du NoC. Les dernières colonnes sont un résumé de leurs résultats d'implémentation.

Tableau 1-1. Caractéristiques des NoC implémentés

NoC	Topologie	Routage	Configuration d'un paquet	Coût du commutateur	Performance maximal	Implémentation
SPIN [ACGM03] [ADGR03] [GUGR00]	Arbre élargi	Déterministique et adaptive	32 bits de données + 4 bits de contrôle	0,24 mm2 CMOS 0,13µm	100 Gbits/s débit cumulé pour réseau	Masque ASIC 4,6 mm2 CMOS 0,13µ
aSOC [JIST00]	Maille	En fonction des applications	32 bits	50 000 transistors	Non disponible	Masque ASIC CMOS 0,35µ
Dally [DATO01]	Maille replié en tore 4x4	En XY selon la source	256 bits de données + 38 bits de contrôle	0,59 mm2 CMOS 0,1µ	4 Gbits/s par commutateur	Non
Octagon [KAND02] [KNDR01]	Anneau *Chordal* avec 8 composantes	Distribué et adaptive	Variable données + 3 bits de contrôle	Non disponible	40 Gbits/s	Masque ASIC
Marescaux [MBVV02]	Maille replié en tore extensible	En XY, bloquant, trou de ver et déterministique	16 bits data + 3 bits de contrôle	446 slices, 4,8 % d'un XCV800	320Mbits/s par canal virtuel à 40 MHz	FPGA Virtex Virtex-II
Aethereal [RGRM03]	Maille 5x5	Trou de ver, tramission garantie, meilleur effort	32 bits	0,26 mm2 CMOS 0,12µm	80 Gbits/s	Masque ASIC
Proteo [ALNU03] [SINU02]	Anneau bidirectionnelle	Non disponible	Variable pour bits de données et de contrôle	Non disponible	Non disponible	Masque ASIC CMOS 0,18µ
Xpipes [BEBE04]	Irrégulière adapté à l'application	Chemins statique en fonction des destinations	32, 64 ou 128 bits	0,33 mm2 CMOS 0,1µ	3 Go/s par commutateur en 32bits @ 500 MHz	Masque ASIC
Hermes [MMMO03]	Maille extensible	En XY, stockage et réémission	Sectionné par segment de 8 bits de données + 2 bits de contrôle (paramétrable)	631LUTs, 316 slices Virtex II	500 Mbits/s par commutateur pour 25 MHz	FPGA Virtex-II
LiPaR [SBKV05]	Maille 1x2, 2x2, 3x3 4x4 max	En XY, arbitre Round-Robin pour les canaux	Sectionné par segment de 8 bits de données + 8 bits de contrôle	772 LUTs, 352 slices 2,57% d'un XC2VP30	33 MHz	FPGA Virtex-II XC2VP30

On constate que les réseaux en maille sont les plus populaires et qu'ils utilisent souvent la méthode de routage XY. Leurs commutateurs a quatre ports de direction (nord, sud, est, ouest) et un ou plusieurs ports locaux pour la composante. La petite taille de configurations des mailles (e.g. 1x2, 2x2, 3x3) est probablement lié au fait qu'une implémentation matérielle nécessite beaucoup de temps et d'argent. D'ailleurs, la plupart des implémentations sur ASIC ne discutent pas des résultats après production de la puce, seulement des simulations sur ordinateurs sont présentées.

La configuration des paquets est particulièrement liée à l'algorithme de routage choisi et des restrictions d'utilisation d'espace. Comme certains NoC dans le tableau, lorsqu'un paquet est divisé en segment (*flit*), cela permet d'utiliser différentes méthodes de routage et réduire considérablement l'utilisation des ressources matérielles. Par exemple, si un paquet de 32 bits transmis par quatre segments de huit bits permet de diviser par un facteur de quatre la taille du paquet, les ressources seront ainsi réduites par le même facteur. En contrepartie, la commutation prendra plus de cycles et augmentera par un facteur encore plus grand la latence d'une communication de la source à la destination.

Plus la taille des paquets est grande, plus large sera la bande passante du réseau. Les problèmes de contention seront réduits et le réseau pourra ainsi être utilisé pour des applications plus lourdes. Évidemment, l'utilisation des ressources matérielles en sera augmentée.

Malheureusement, les coûts et résultats de performance sont difficilement comparables. La taille des paquets, le nombre de composantes pouvant être connecté au NoC et la technologie choisie rendent difficiles la comparaison de résultats. Par contre, certaines tendances ressortent :

- La bande passante des ASIC se situe au niveau Gb/s (*Gigabit par seconde*), alors que la bande passante des FPGA se situe au niveau Mb/s (*Mégabit par seconde*)
- La fréquence de fonctionnement des NoC ASIC est dans les centaines de MHz (*Mégahertz*) et celle des FPGA dans les dizaines de MHz
- Il y a autant d'implémentations ASIC que FPGA. Le coût et les performances d'un commutateur pour une même technologie sont très semblables.

Une seule des architectures présentées a une structure hétérogène (xpipes). Ce réseau est grandement paramétrable et a été développé en SystemC. Il a la particularité d'avoir des commutateurs disposés de façon irrégulière et pouvant s'adapter aux besoins des applications. Si deux blocs matériels nécessitent beaucoup de bandes passantes, un commutateur sera instancié entre les deux. D'une autre façon si la bande passante n'est pas importante, les coûts seront réduits en connectant ces blocs sur un commutateur déjà existant.

Pour tous les NoCs présentés, aucun n'a la topologie crossbar et ou ne fonctionne pas selon des communications non bloquantes. Un réseau personnalisé a la possibilité d'être configuré non bloquant s'il utilise des configurations classiques de commutation (*clos, benes*) [FENG81]. Évidemment, être non bloquant n'est pas la propriété obligatoire pour qu'un réseau soit performant. Ce sont les requis qui déterminent si un réseau est considéré performant. Par exemple, assurer la qualité des communications (en anglais, *Quality of Services*, QoS) pour le NoC Aethereal [RGRM03] est une propriété considérée importante.

1.3.1 Produits commerciaux

Récemment, différents produits de haute performance incorporant un NoC sont apparus sur le marché. À l'intérieur de ces produits, plusieurs coprocesseurs et mémoires sont interconnectés par un NoC et réalisent d'énormes quantités de calculs en parallèle. Les informations de ces architectures étant gardées secrètes, de brèves explications et figures de ces NoC sont fournies dans cette section.

1.3.1.1 Radeon

Le nouveau processeur graphique des cartes Radeon de l'entreprise ATI [ATI00] utilise un bus en anneau bidirectionnel de 128 bits pour accéder ses huit mémoires externes. Pour réduire la latence et simplifier l'implémentation du réseau, son placement est limité à l'entour de la puce. Les IP sont au milieu de la puce avec le contrôleur de mémoire (arbitre des bus) au centre. Le contrôleur est encerclé d'un crossbar pour un accès rapide au IP qui fait des requêtes d'accès aux bus. Les seuls points d'interface pour accéder les mémoires sont situés à quatre arrêts d'anneaux (*ring stops*). Avec une technologie de 0,09 µm (micromètre) et une fréquence de 625 MHz, le bus peut atteindre un débit maximal de 48 Go/s (*Gigaoctet par seconde*).

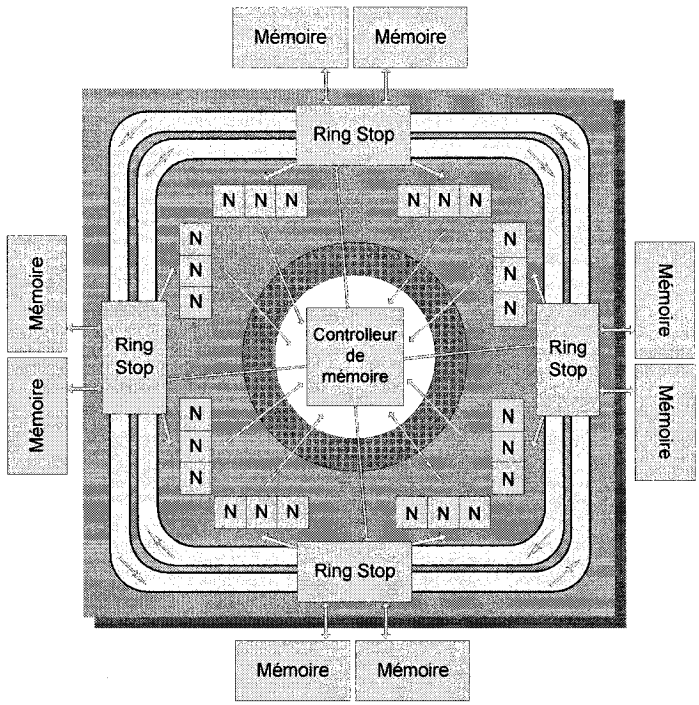


Figure 1-8. Modèle pour les accès mémoires de la puce graphique Radeon

1.3.1.2 CELL

Le très populaire processeur CELL [IBM00], présenté à la **Figure 1-9**, est considéré comme la réalisation d'un superordinateur sur une puce. Développé par IBM, Toshiba et Sony, il comporte un processeur PowerPC (PPE) principal, huit coprocesseurs fonctionnent en synergie (*synergistic processor unit*, SPU ou SPE) et des ports de communication externe. L'interconnexion se fait par l'entremise de deux bus de communication en anneaux bidirectionnels. Ce bus d'interconnexion pour les éléments de calculs (*element interconnect* bus, EIB) a 128 bits de large. À chaque transaction, le réseau peut transmettre 128 octets. Ces transactions volumineuses sont nécessaires pour effectuer d'impressionnants calculs en parallèle. La gestion des communications se fait par un arbitre situé au milieu du EIB. Les transactions prennent trois cycles pour passer d'un bloc SPU à un SPU voisin. L'utilisation d'un crossbar au lieu du EIB avait été envisagée au début de la conception du processeur, mais après révision, le crossbar prenait trop d'espace sur la puce.

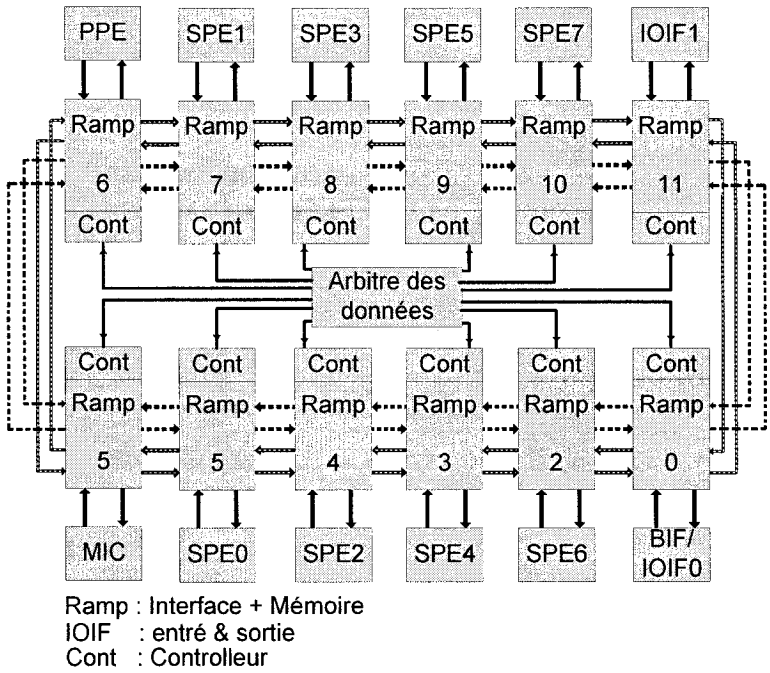


Figure 1-9. Modèle des bus de communication en anneaux du processeur CELL

1.3.1.3 ClearConnect

ClearConnect® est un réseau sur puce disponible comme solution IP [CLEA01]. Son architecture s'adapte spécifiquement aux applications (comme xpipes). Ses commutateurs, disposés irrégulièrement, sont interconnectés en chaînes (**Figure 1-10**). L'ajout de commutateurs dans la chaîne augmente la bande passante. Développé par l'entreprise ClearSpeed™, ce réseau a des chemins réduits par son architecture adaptée et ne dissipe aucune puissance s'il n'y a pas de transfert de donné. Ces deux facteurs permettent de réduire la consommation de puissance. L'entreprise annonce des débits de 12,5 Gb/s pour un bus de 128 bits à 400 MHz. Le réseau ClearConnect® est d'ailleurs utilisé dans le processeur (SoC) CSX600 de l'entreprise. Les possibilités d'extensibilités de ce réseau sont très grandes. Le processeur CSX600 comporte jusqu'à 96 éléments de calcul interconnectés pour effectuer des calculs en virgule flottante.

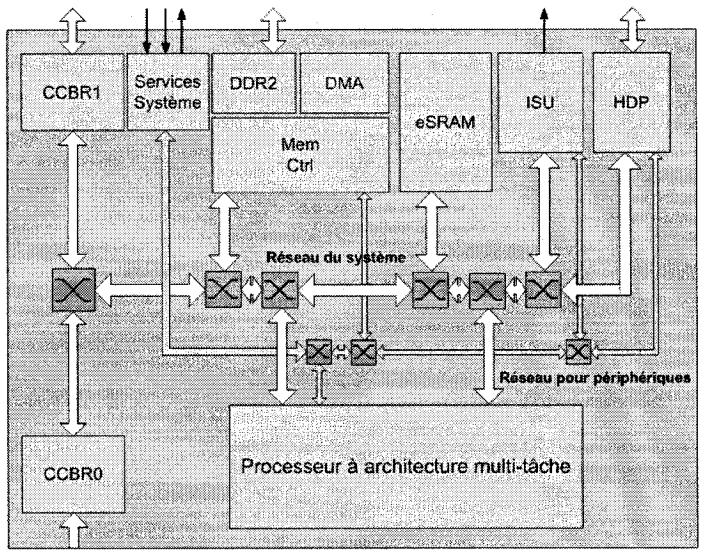


Figure 1-10. Réseau ClearConnect® utilisé dans le processeur CSX600 de Clearspeed

1.3.1.4 STBus & STNoC

Le STBus [STMI00] est une architecture réseau intégrée sur puce modulaire et développée par l'entreprise STMicroelectronics. Chaque nœud est considéré comme une matrice de commutation. Différentes configurations de ces nœuds peuvent être instanciées pour donner une structure d'interconnexions hiérarchique. Le nœud peut être un bus partagé, un crossbar ou un crossbar partiel. Le crossbar partiel est un compromis entre un bus peu coûteux, mais lent et un crossbar très performant, mais coûteux en fils. Le STBus utilise trois protocoles de communication pour diverses catégories de composantes : périphérique, de base et avancé. Il incorpore des convertisseurs de protocoles ou pont (en anglais, *bridge*) de communication, tampons de synchronisation et convertisseur de taille de paquets pour établir une communication entre des blocs IP utilisant différents protocoles. Il supporte le QoS pour différencier la priorité des paquets et assurer que tout paquet atteint sa destination sans erreur.

STMicroelectronics a poussé son concept de réseau sur puce encore plus loin pour répondre aux problèmes de complexités des SoC. Avec sa technologie STNoC™

[STMI00], les outils fournis pour les clients ASIC de STMicroelectronics sont améliorés. La nouvelle architecture *spidergon* (**Figure 1-11a**) est un anneau *chordal* avec des interconnexions entre voisins complètement opposés pour obtenir une latence moindre que le STBus. Compatible avec de nouveaux protocoles comme AXI (*Advanced eXtensible Interface*) [ARM01] et OCP (*Open Core Protocol*) [OCPI00]. Il supporte la connexion à des réseaux STBus. Un point intéressant de la topologie *spidergon*, est que son nombre de ports est moindre que la topologie en maille. Son implémentation planaire est simplifiée (**Figure 1-11b**), sa complexité de routage est diminuée et son coût est réduit.

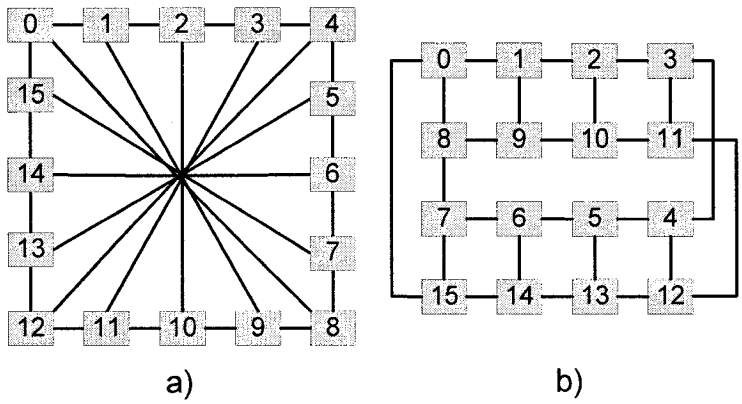


Figure 1-11. a) Modèle NoC du *spidergon* b) Implémentation planaire à 16 nœuds du *spidergon*

1.3.1.5 Arteris

Arteris [ARTE06], présenté à la **Figure 1-12**, offre un NoC avec une topologie en maille extensible. Des outils sont fournis pour intégrer aisément le bloc IP dans les flots de conception existants. Le réseau supporte plusieurs protocoles de communication à ses interfaces et prend avantage du concept de globalement asynchrone et localement synchrone. Cela permet de diminuer les restrictions d'utilisation de fréquences de fonctionnement différentes. Ainsi, les modules du SoC peuvent opérer à leur propre fréquence de fonctionnement maximale et ainsi améliorer la performance globale du SoC.

Le NoC peut atteindre une fréquence impressionnante de 750 MHz. Inspiré du modèle OSI (*open systems interconnection*), le réseau est structuré de trois couches de communication. Cela permet de facilement faire le lien en une couche et les critères permettant l'amélioration des performances. La couche de transaction identifie les besoins en mémoire et les protocoles de communication. La couche de transport définit le routage des paquets et permet d'optimiser selon les besoins d'une application. Par exemple, un routage en trou de ver permet de diminuer la latence et réduire le besoin en mémoire. Finalement, la couche physique est influencée par les caractéristiques de la technologie de semi-conducteur utilisé.

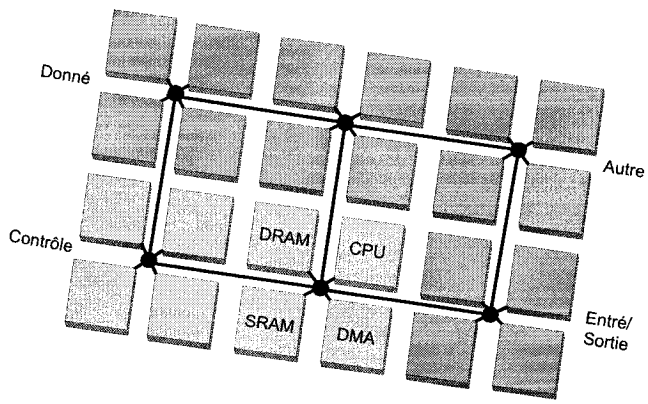


Figure 1-12. Architecture en maille du NoC Arteris

1.4 Conclusion

Les architectures de NoC présentées sont diversifiées et promettent toutes d'accroître les performances d'un SoC. On peut remarquer une prédominance des topologies en bus et en anneaux. Certaines architectures ont une méthode de routage complexe et adaptée aux applications. En général, elles utilisent peu d'espace sur la puce, offrent des latences intéressantes et leur débit de communication est grandement meilleur que les anciennes topologies en bus. Étant implémentée sur ASIC, leur fréquence de fonctionnement est généralement très grande.

La maille est davantage utilisée dans le domaine de la recherche. Elle est facilement extensible et semble être une solution axée sur les futurs SoC incorporant plus d'une dizaine de composantes. Présentement, ce n'est pas le cas pour les produits commerciaux. Ceci est probablement du au fait que les bus et les anneaux sont des architectures maîtrisées depuis longtemps, peu d'entreprises utilisent la maille.

Plusieurs entreprises qui ont fait le choix des NoC sont des joueurs importants du domaine des semi-conducteurs. On peut affirmer que les NoC ne sont plus qu'un simple paradigme, mais un concept architectural permettant de résoudre les problèmes de complexité des SoC.

CHAPITRE 2

LE RESEAU SUR PUCE ROTATOR ON CHIP

Ce chapitre présente le RoC (Rotator on Chip), un réseau intégré sur puce développé en partenariat avec STMicroelectronics de Ottawa. Simple et extensible, le RoC est un modèle de NoC utilisant une topologie en anneau. Peu de réseaux trouvés dans la littérature utilisent un modèle comme celui-ci. Par contre, le RoC comporte plusieurs fonctionnalités de base présentes dans les NoC dont celle de permettre l'échange de données par paquets pour les composantes connectées au réseau. Les premiers développements du réseau ont été programmés en C++ et avec la librairie SystemC [DESL05]. Le fait de développer avec un niveau d'abstraction plus élevé permet d'évaluer les fonctionnalités du RoC et déterminer quels sont les paramètres qui améliorent les performances.

Comme le modèle haut niveau, le modèle bas niveau est extensible pour le nombre de composantes connectées au réseau, la taille des paquets, source, destination et des données. Pour évaluer les performances réelles du modèle, une implémentation matérielle (implémentation bas niveau du modèle) est nécessaire et sera vue dans le prochain chapitre. Pour débuter, une explication des modules du modèle de base est expliquée et différentes versions du modèle sont présentées. Par la suite, l'ajout de certaines fonctionnalités aux modèles est discuté, permettant ainsi au réseau d'atteindre de meilleures performances.

2.1 Vue d'ensemble

Basé sur le modèle réseau de l'anneau par jeton (en anglais, *Token Ring*) et inspiré des architectures de commutateurs en télécommunication, il permet d'établir un lien de communication entre plusieurs composantes. Ces composantes sont connectées à l'entour

de l'anneau à chacune des paires d'entrée et sortie. Normalement, pour qu'un client connecté à un réseau *Token Ring* puisse communiquer, il doit obtenir un jeton pour transmettre des messages. Différemment, le RoC permet que toutes composantes transmettent au réseau un paquet lorsqu'elles le requièrent. De plus, son architecture peut être configurée non bloquante [FENG81] comme un crossbar et ainsi supporter simultanément n'importe quelle communication entre une source et une destination.

Pour le RoC, le terme ressource est utilisé et représente une ou des composantes dans un seul bloc faisant appel au réseau comme moyen de communication. Le RoC permet la connexion de N ressources par l'entremise de ses interfaces de communication. Son modèle est divisé en deux parties, soit le nœud et la banque. Chacun des nœuds incorpore une interface pour l'échange de données et peut supporter différents protocoles de communications. De façon générale, le transport d'un message d'une ressource à une autre commence lorsqu'un nœud reçoit une requête de la ressource source. Le nœud transforme cette requête en paquet et transmet le paquet à une banque. Les banques effectuent une rotation en sens horaire et acheminent ainsi le paquet vers son nœud de destination. Finalement, le paquet sort du RoC sous forme d'une requête par l'interface du nœud de destination et permet ainsi d'atteindre la ressource destination. Plus le processus est transparent, plus simple sera l'intégration du réseau à un système. Dans la **Figure 2-1**, le modèle de base du RoC est présenté. Dans l'exemple, quatre ressources (N = 4) sont connectées au réseau et cinq paquets identifiés d'une adresse de source et de destination sont prêts à être transmis à leur banque.

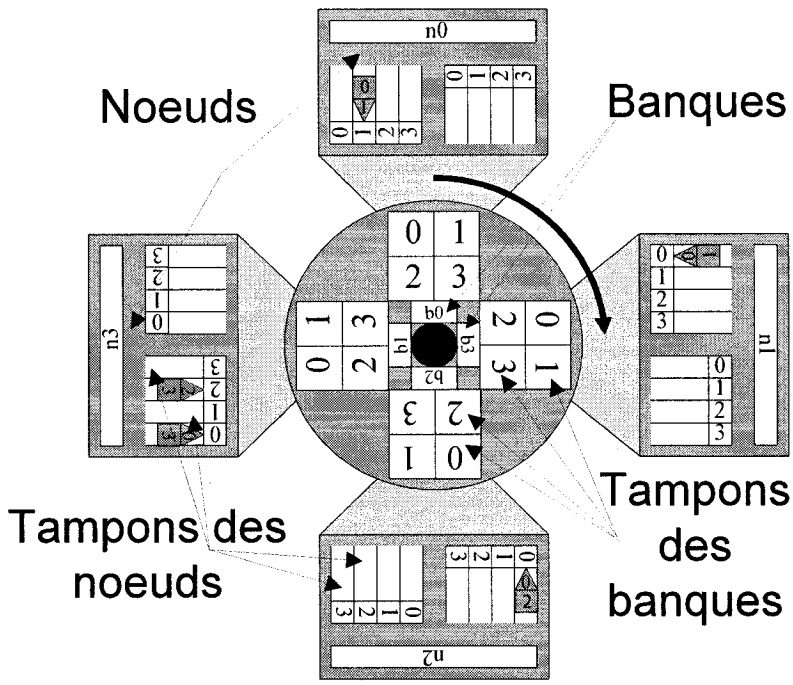


Figure 2-1. Architecture haut niveau du RoC

2.2 Le noeud

Il sert principalement à transformer les requêtes/réponses des ressources en paquets (ou segments de messages) à acheminer dans le réseau et temporiser les échange de paquets avec la banque. Chaque ressource connectée à une interface peut communiquer avec N-1 ressources. L'interface première du RoC utilise un ou des FIFOs pour les entrées et sorties, plusieurs configurations sont possibles. Une ressource qui s'interface avec un FIFO simplifie la connexion au RoC. Par contre, si la ressource utilise un protocole de communication particulier, une interface supplémentaire (appelé enveloppe) est nécessaire et se superpose à l'interface des FIFOs. Chaque nœud est composé de N - 1 tampons comme entrée FIFO (en fonction de la destination) et N - 1 tampons comme sortie FIFO (en fonction de la source). Les paquets comportent principalement trois champs d'information nécessaires au transport des données, mais rien n'empêche d'ajouter des champs supplémentaires pour des raisons de fonctionnalités. Les champs

sont: adresse source associée au nœud où le paquet entre dans le réseau, adresse de destination associée au nœud où le paquet sortira du réseau et le champ des données. Les paquets sont de taille fixe et se font estamper d'une adresse source et d'une adresse de destination par l'interface. Par la suite, ils sont temporisés dans un FIFO en fonction de sa destination.

À chaque fois que le noeud veut transmettre des données, il doit examiner l'état de ses FIFOs d'entrées et construire une demande de transfert des paquets en fonction des priorités. La version de base du RoC utilise un algorithme d'alternance de permissions. Cet algorithme est utilisé, car il est simple, mais d'autres algorithmes plus complexes peuvent être utilisés pour améliorer les performances du réseau (exemple : en fonction de paramètres QoS insérés dans les paquets, avantager certaines destinations). Lorsqu'une permission est établie, le nœud fait une requête (inclut l'information de la destination voulue) à la banque pour transmettre un paquet. Si la requête est acceptée, à la prochaine étape (lorsque le noyau aura pivoté d'un coup) le paquet est transmis à la banque, sinon aucun transfert n'est effectué.

2.3 La banque

Il existe N banques (b0 à b3 dans la **Figure 2-1**) et son ensemble est considéré comme le cœur (ou noyau) du réseau. Chaque banque est composée de N tampons (0 à N-1). Ainsi, le nombre requis de tampons est N^2. La banque joue le rôle de commutateur dans le réseau et permet que les paquets voyagent de la source à la destination. Chaque tampon dans une banque est associé à un numéro d'identification correspondant à un nœud de destination. Cet identifiant permet d'associer un tampon à l'adresse de destination d'un paquet. Le cœur effectue une rotation sur lui même dans le sens horaire, il change ses connexions aux nœuds et effectue une rotation complète (360 degrés) en N étapes. Le fait de déplacer les paquets d'une banque à l'autre ou que toutes les banques effectuent une

rotation en même temps revient au même. Tous deux permettent aux paquets d'être acheminés à leur destination.

Durant chaque étape, le mécanisme du RoC permet de router les paquets. Une banque est connectée avec exactement un seul nœud. Le nœud peut envoyer un paquet à sa banque seulement si le tampon correspondant de la banque est libre. Le tampon de banque où le paquet sera temporisé doit correspondre à l'identifiant de destination du paquet. Un seul paquet est transmis du nœud à sa banque à chaque pivot. Durant cette même étape, un paquet ayant atteint son nœud de destination sortira de son tampon de banque en étant transmis au tampon de nœud correspondant avec son identifiant de source. Par la suite le paquet quitte le réseau par l'interface du nœud. Étant donné que le chemin pour atteindre une destination est toujours le même et qu'un seul paquet peut être déposé dans un tampon de banque, l'ordre des paquets arrivant à un nœud est toujours garanti. Tout paquet partant du même nœud ne peut pas dépasser un autre paquet pour arriver à la même destination. Finalement, chacune des spécifications mentionnées dans ce paragraphe fait partie d'une étape de rotation. Une étape peut comprendre plusieurs sous étapes de fonctionnements (décodage, établissement du chemin des données, temporisation, etc.).

Pour simplifier le modèle du RoC, la notion de canal de communication est utilisée pour expliquer le chemin fixe qu'empruntent les paquets au moment où ils quittent le nœud pour entrer dans une banque et jusqu'à leur sortie vers le nœud de destination. Le canal qu'emprunte un paquet est choisi en fonction de son adresse de destination. Le paquet finit son parcours dans le canal lorsqu'il atteint le tampon du nœud de destination correspondant à sa source. Il ne pourra prendre d'autres avenues, seul un délai peut survenir du à l'attente dans le FIFO d'entrée.

2.4 Variantes du modèle

Les modules détaillés précédemment concernent le fonctionnement général du RoC. Dans cette section, différentes variantes au fonctionnement du RoC seront abordées. Ces fonctionnalités permettent un gain important des performances et peuvent être incorporés aux différents modèles du RoC. En effet, un modèle peut utiliser une combinaison de ces fonctionnalités pour obtenir davantage de gain. Différentes configurations sont ainsi possibles pour répondre aux besoins, mais dépendamment de la configuration, des modifications à l'architecture du RoC seront nécessaires pour permettre l'intégration des fonctionnalités. Certaines de ces versions ont été réalisées et leurs résultats sont disponibles dans les prochains chapitres ou dans l'ancien mémoire concernant le RoC [DESL05].

2.4.1 Version multiplexeur avec banque

Par l'entremise de multiplexeurs qui font office de commutateurs, le contrôleur utilise des multiplexeurs pour gérer le chemin qu'empruntent les paquets à travers le réseau. Ces multiplexeurs font office de commutateur et permettent de router les paquets du nœud vers la banque et échanger banque à banque. Le RoC haut niveau [DESL05] est orienté vers ce type d'architecture, mais différentes variantes de ce modèle sont possibles.

2.4.1.1 Version du commutateur à N cycles

Cette version est axée sur implémentation matérielle possible du modèle *multiplexeur* avec *banque*. Présenté à la **Figure 2-2**, ses canaux de communication sont réorientés à chaque rotation et nécessitent l'utilisation de gros multiplexeurs pour commuter les paquets entre les nœuds et les banques. Deux blocs de commutateurs regroupent tous les multiplexeurs. Le premier bloc permet l'échange des données entre les tampons de nœud et les tampons de banque. Le second termine le chemin échangeant les données entre les tampons de banque et les tampons de nœud de la destination. Au lieu d'avoir N tampons

de banques par banque, donc N^2 tampons de banques, seulement un tampon par banque est nécessaire. Le tampon doit préférablement être un FIFO d'une profondeur permettant aux paquets de s'empiler avant leur distribution au nœud destinataire. Si le FIFO n'est pas assez grand, l'espace n'est pas disponible et le nœud d'entrée refuse l'écriture au FIFO. Ce modèle réduit grandement l'utilisation de mémoires, mais risque d'occuper beaucoup d'espace avec ses multiplexeurs. Le réseau permet tout de même d'obtenir une complexité matérielle intéressante de O(N log N).

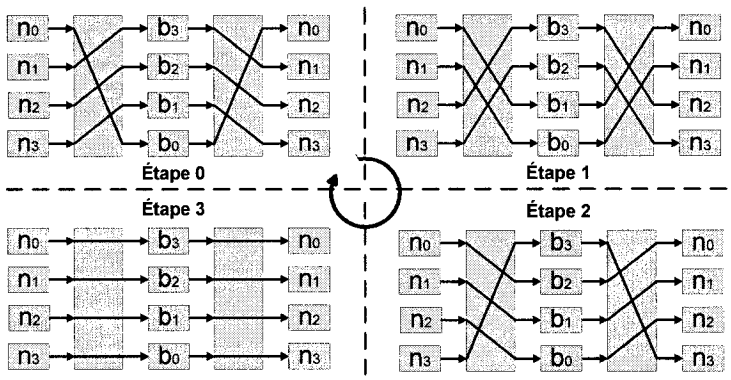


Figure 2-2. Fonctionnement du RoC pour implémenter un commutateur à N cycles

2.4.2 Version registres à décalage

Inspiré de l'architecture du registre à décalage (en anglais, *shift register*), cette version simplifie la commutation entre le nœud et la banque. Un multiplexeur (**Figure 2-3**) à l'entrée de chaque mémoire permet de sélectionner entre les paquets provenant de la mémoire précédente (tampon de banque) ou insérer un nouveau paquet dans le canal de communication (provenant du tampon de nœud). Un paquet déjà dans le canal se déplace au prochain registre à chaque rotation. La simplicité de son architecture permet de facilement évaluer son rendement (latence, problème de contention) et ainsi évaluer ses capacités de communication en fonction. De plus, l'ajout de fonctionnalités comme le QoS ou la réorganisation de ses canaux entre chaque nœud requiert peu de modifications.

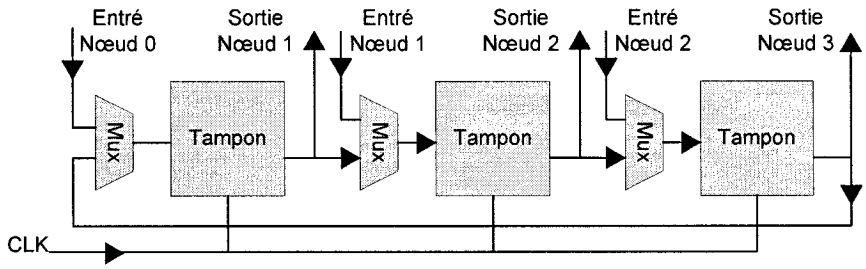


Figure 2-3. Canal de communication inspiré du registre à décalage

2.4.3 Version HyRoC

Ce modèle conçu et breveté [DESL05] par STMicroelectronics de Ottawa est une extension du RoC de base. L'architecture du HyRoC (*Hyper Ring on Chip*) inclus des canaux de communication en anneau parallèle pour minimiser l'interblocage. Ses anneaux à deux dimensions permettent d'accroître l'extensibilité du réseau et minimiser les latences. Disposé en coordonné X et Y (semblable aux mailles), ses cellules sont constituées des anneaux, des interfaces et des ressources. Son mécanisme de rotation peut être physiquement implémenté en utilisant des registres à décalages ou des commutateurs. Son architecture partage des caractéristiques similaires à la maille en *tore* et au réseau ClearConnect®. La différence principale entre sa topologie et celle en *tore* réside du fait que les paquets ne commutent pas à chaque cellule, mais effectuent parfois des sauts à la prochaine cellule (**Figure 2-4**). En utilisant un multiplexeur, le paquet est tout simplement déposé ou enlevé du canal de communication. Ainsi, nul besoin de temporiser les paquets en transit vers le canal. L'espace sauvé en éliminant les tampons permet d'utiliser de multiples canaux de communication parallèle. Un modèle en SystemC a été développé de cette version pour évaluer son comportement face au trafic [DELI05]. Dans les figures qui suivent, un exemple à 16 cellules est présenté avec une configuration à deux canaux. Un paquet partant du commutateur C0,0 et qui veut atteindre C2,2 a plusieurs possibilités de chemins. Il peut par exemple débuter dans l'anneau H1,0 et se déplacer jusqu'à atteindre l'anneau V0,2. L'anneau V0,2 permet

d'atteindre le commutateur C2,2. L'ensemble du parcours nécessite donc quatre déplacements.

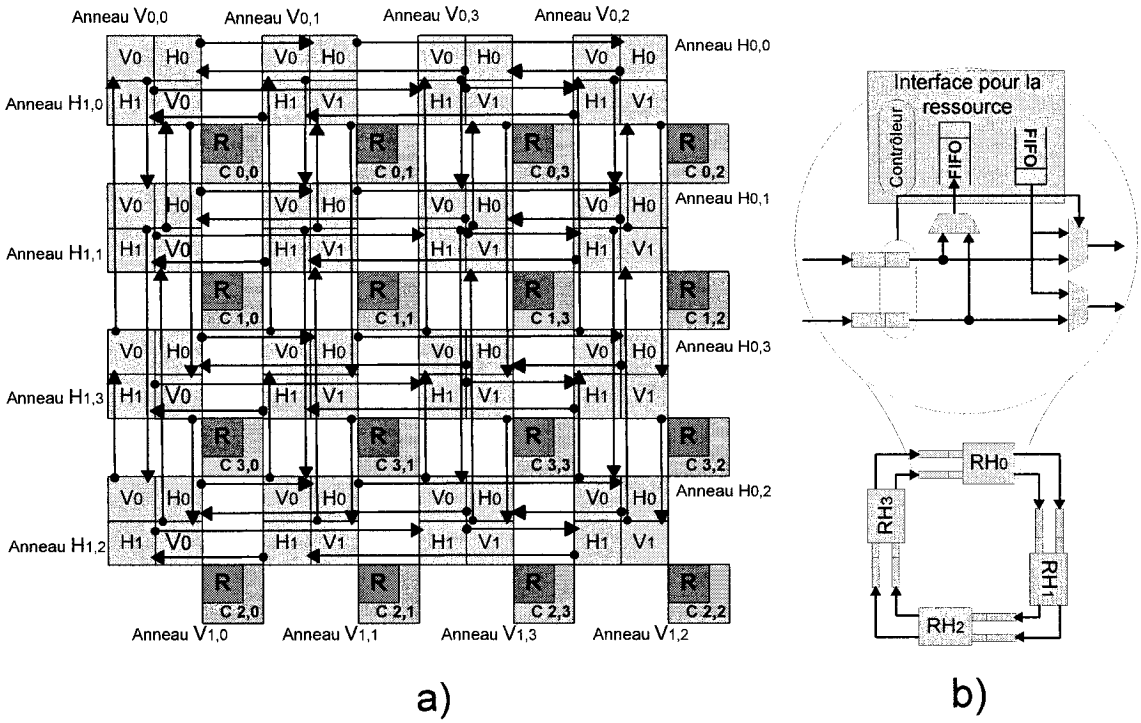


Figure 2-4. a) Topologie HyRoC à deux dimensions b) Architecture générale des anneaux

2.5 Fonctionnalités et optimisations architecturales

L'ajout de fonctionnalités et d'optimisations à l'architecture de base du RoC lui permet d'obtenir de meilleures performances sur le plan de l'utilisation des ressources ou du débit de sa bande passante. Elle permet également de réduire le délai de transmission (latence) des paquets de la source à la destination. Certains coûts peuvent y être associés : augmentation de la complexité (e.g. latence de communication et du matériel) et temps de conception. De plus, certaines étapes peuvent comporter des sous étapes et nécessiter ainsi l'ajout de tampons pour temporiser les paquets. Dépendamment du modèle, des fonctionnalités et des optimisations choisies, ceux-ci peuvent influencer la méthode de

routage et ainsi imposer des changements à l'architecture. Certains des ajouts ont été évalués dans le modèle haut niveau du RoC [DESL05] et ont permis d'observer un gain important des performances.

2.5.1 Matrice des mémoires

Aussi appelée en anglais *bitmap*, cette fonctionnalité peu complexe permet de garder dans une matrice un relevé d'utilisation des mémoires. Sachant quelle mémoire est remplie ou vide, cette information permet de rapidement mettre à jour le chemin de données à chaque étape. Lorsqu'une mémoire reçoit un paquet, le *bitmap* associé à cette mémoire est inscrit d'une valeur (1 pour utilisé et 0 pour une mémoire inutilisé). Et à la fin, lorsque le paquet quitte une mémoire pour atteindre une autre, le *bitmap* est effacé. Sachant que la consommation de puissance dans un circuit est principalement liée aux transitions des bits, le fait d'effacer un bitmap au lieu d'une mémoire entière permet de minimiser la consommation. Cette méthode rapide et simple ne requiert aucun décodage des mémoires pour déterminer si un paquet est présent. Notez que sans matrice des mémoires, il faut vider la mémoire, car à chaque cycle les données sont routées automatiquement dans le réseau. Il n'y a pas de méthode pour savoir si un paquet est présent dans une mémoire.

Dans la matrice, une partie des bitmaps est associée pour chaque mémoire. Dans la **Figure 2-5**, l'exemple montre que l'envoi des paquets d'un nœud vers sa banque nécessite que la banque sache quels tampons du nœud sont remplis et lesquels de ses tampons de banques sont vides pour recevoir les données. Cette méthode permet de réduire le besoin en communication entre le nœud et la banque. La banque détermine quels paquets elle veut recevoir et au cycle suivant les reçoit automatiquement. On peut aussi utiliser cette méthode de matrice pour les tampons de banques. Lors des échanges banque à banque, la matrice des tampons peut être échangée entre celles-ci pour sauver le nombre de processus à effectuer et ainsi accélérer le transfert.

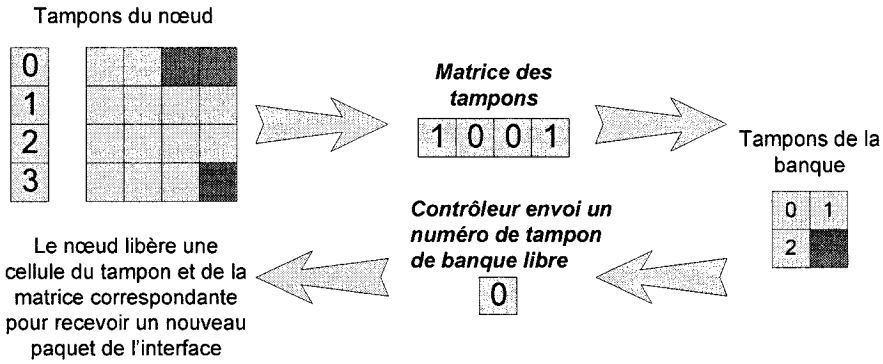


Figure 2-5. Envoie de la matrice des tampons et réponse du paquet à recevoir

2.5.2 Fonctionnement bidirectionnel

Cette fonctionnalité, implémentée dans le modèle de base du RoC en SystemC [DESL05], permet d'offrir des gains en performance très intéressants. Les déplacements de données dans ses canaux de communications se font selon deux directions, horaire et antihoraire (**Figure 2-6**). La plus simple implémentation de cette fonction est d'utiliser seulement deux canaux de communication pour des directions opposées. Un paquet entrant dans une banque est routé dans le canal permettant la plus courte distance (le moins de saut entre les banques) pour atteindre son nœud de destination. Ainsi, par rapport au modèle de base du RoC, le chemin le plus long pour atteindre un nœud de destination est divisé par deux (la latence maximale est donc divisée par deux). La latence moyenne des communications est également divisée par deux, car certains protocoles nécessitent qu'un accusé de réception ou que des données soient retournés vers la source. De cette manière, le paquet ne parcourt jamais toutes les banques pour atteindre une destination et retourner à sa source. Pouvant être implémentée dans différents modèles, cette fonction ajoute peu de complexité à l'architecture. Le contrôle du routage pour le meilleur chemin est le seul élément complexifié. Évidement, pour augmenter la bande passante, une optimisation possible est d'utiliser plusieurs canaux bidirectionnels.

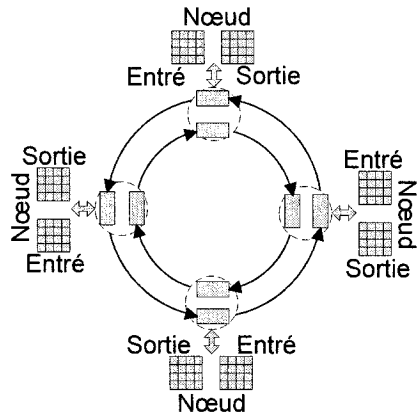


Figure 2-6. Architecture haut niveau du RoC bidirectionnel

2.5.3 Structure hiérarchique

En offrant au réseau différents niveaux de hiérarchisation, différentes configurations sont possibles (**Figure 2-7**) et permettent des gains en performance. Au lieu d'utiliser un très grand RoC, on peut utiliser plusieurs petits sous-réseaux et limiter la latence des communications d'un nœud à l'autre. Certaines ressources n'ont pas besoin d'un accès fréquent au réseau et peuvent être isolé vers un sous-réseau. Ainsi, le centre du réseau est réservé aux ressources plus demandant mais sans perdre accès aux plus petites ressources. Même si cette fonctionnalité permet une panoplie de configurations possibles, elle ajoute de la complexité à la conception du réseau et la méthode de routage est alourdie.

Quel point d'accès au réseau sert comme pont vers d'autres sous-réseaux? Combien de points d'accès (ponts) par réseau? L'ajout de plusieurs sous-réseaux fait apparaître de nouvelles topologies, choisit-on un anneau, un hypercube, une topologie adaptée? Quelle méthode de routage doit-on adopter pour éviter les problèmes de contention? Différentes explorations de la configuration du réseau sont possibles et permettent d'ajuster l'architecture pour chaque application. Un choix judicieux du placement des ressources dans un réseau hiérarchique permet de garder la plupart des transactions dans l'anneau

local. Ainsi, le besoin de transiter les paquets par le centre du réseau est limité. Cela permet de désengorger le réseau, augmenter la bande passante et diminuer la latence des transactions. Une configuration hiérarchique permet de sauver sur le nombre de ressources matérielles, car sa complexité devient O(N). Cette analyse de comparaison entre un RoC simple (un étage seulement) et un RoC hiérarchique a été fait avec le modèle haut niveau [DESL05]. Il devient ainsi moins coûteux en tampons, car instancier plusieurs fois le même réseau permet d'additionner le nombre de tampons au lieu d'en ajouter selon une complexité $O(N^2)$.

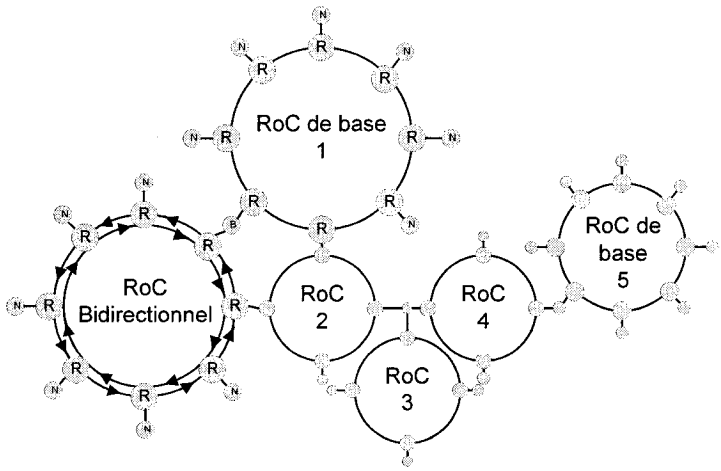


Figure 2-7. Exemple d'un réseau hiérarchique aggloméré de différentes configurations de RoC

2.5.4 Mode rafale

Transmettre de grosse quantité de données dans le réseau nécessite un grand débit de communication. Le débit dépend de la largeur des paquets et du nombre de paquets pouvant être transmis à chaque cycle. Plus un paquet est grand, plus l'espace utilisé sera grand. Ainsi, si on veut augmenter le débit sans utiliser plus d'espace, on doit jouer sur le facteur paquets par cycle. Pour qu'un paquet soit transmis, il est nécessaire d'avoir une requête, une réponse, un transfert et une confirmation. Si N paquets sont transmis, 4N communications sont nécessaires. En utilisant le mode en rafale (en anglais, *burst mode*), on abaisse ce nombre à N + 3, soit une requête, une réponse, N transferts et une

confirmation. Ce mode permet un gain important des performances, mais comporte son lot de complexité.

Il implique qu'une suite successive de paquets soit transmise dans un canal de communication totalement réservée pour le transfert. Aucun paquet ne peut entrer dans le canal avant que le mode rafale soit terminé. Ceci complexifie son intégration et peut occasionner des goulots d'étranglement. Si le mode rafale bloque un canal de communication trop longtemps, certains nœuds pourront se voir refuser l'accès à de multiples reprises et ainsi compromettre les performances du réseau. Pour éviter une telle situation, il faut que le réseau garde en mémoire les paquets qui ne seront pas transmis et les envoient une fois que le transfert en rafale sera terminé. Tel que discuté à la section 2.1, les nœuds comportent des tampons pour attendre que les transferts refusés aux banques soient acceptés. Toutefois, si le trafic est lourd et que le transfert en rafale interrompt constamment les communications, des tampons supplémentaires sont nécessaires et ajoutent ainsi un coût en espace.

2.5.5 Élimination de la moitié des tampons

La complexité du RoC est importante et différentes méthodes sont nécessaires pour réduire son utilisation en matériel. La configuration hiérarchique n'est pas la seule. Par simulation avec le modèle haut niveau du RoC [DESL05], il a été démontré que le taux d'utilisation des tampons est d'environ 50% lorsqu'il incorpore la fonctionnalité des matrices. Ces résultats laissent présager qu'il est possible de réduire le nombre de tampons sans dégrader les performances. En fait, le résultat des simulations a démontré que la latence des paquets est peu affectée lorsque seulement la moitié des tampons sont présents. Cette optimisation affecte le fonctionnement. En effet, les paquets sont alors routés lorsqu'un canal de communication devient libre. Ce n'est donc plus un seul canal par destination comme pour le modèle de base, mais plutôt N/2 canaux pour N

destinations. Ainsi, le contrôleur des banques doit être modifié. Même si la réduction des tampons modifie grandement un modèle et y ajoute de la complexité, le gain en espace est important et mérite d'être considéré lors d'une implémentation matérielle.

CHAPITRE 3

IMPLEMENTATION MATERIELLE

Il existe au moins deux architectures pour réaliser physiquement le noyau du RoC. La première méthode a été présentée à la section 2.4.1.1. Le mécanisme de rotation pour cette méthode fait usage de deux blocs de commutation qui alterne les connexions entre les nœuds et les banques. Les banques peuvent être réalisées comme des mémoires distribués avec de simples contrôleurs qui chargent et libèrent les paquets. Il se peut que l'implémentation matérielle de ces blocs de commutation ne soit pas régulière et génère de longs chemins de délais de propagation qui affecte la latence du réseau. La complexité de la communication est O(N log N) et nécessite seulement N permutation pour connecter les nœuds et les banques. Par contre, cette implémentation est bien moins complexe qu'un crossbar (section 1.1.1.2.).

La seconde approche basée sur les registres à décalage implémente le mécanisme de rotation en décalant l'information d'une banque à l'autre. Chaque banque est à cheval sur un nœud et cette disposition permet d'être très régulière. En revanche, la quantité d'information à décaler peut s'avérer importante. Une façon d'y remédier, telle que vue dans la section 2.5.5, est de réduire la quantité de tampons de banque sans qu'un impact significatif ne soit perçu sur les performances. À un autre extrême, utiliser seulement un tampon par banque est une solution possible, mais l'architecture devient similaire à un simple bus ClearConnect® [CLEA01].

L'implémentation du RoC que nous avons réalisée est axée sur cette deuxième méthode. La **Figure 3-1** représente le RoC sous sa version à quatre nœuds. Au lieu d'utiliser un FIFO pour chaque destination comme dans le modèle de base, un seul FIFO est utilisé et est combiné à des tampons pour chaque destination. Dans la **Figure 3-1**, les tampons sont

représentés par les carrés (exemple : les registres à décalage). Cette configuration permet au RoC de générer des communications non bloquantes et améliore ainsi l'analyse de sa complexité de fonctionnement. Il est à noter que cette configuration n'est pas l'optimisation vue en 2.5.5.

Les prochaines sections présentent en détail l'implémentation du modèle bas niveau avec les modifications apportées par rapport à son modèle de base. Les discussions portent principalement sur le mécanisme, les paramètres de configurations et les méthodes de vérification du design. Pour obtenir de plus amples informations sur l'architecture et son fonctionnement, le lecteur peut consulter le code VHDL en annexe.

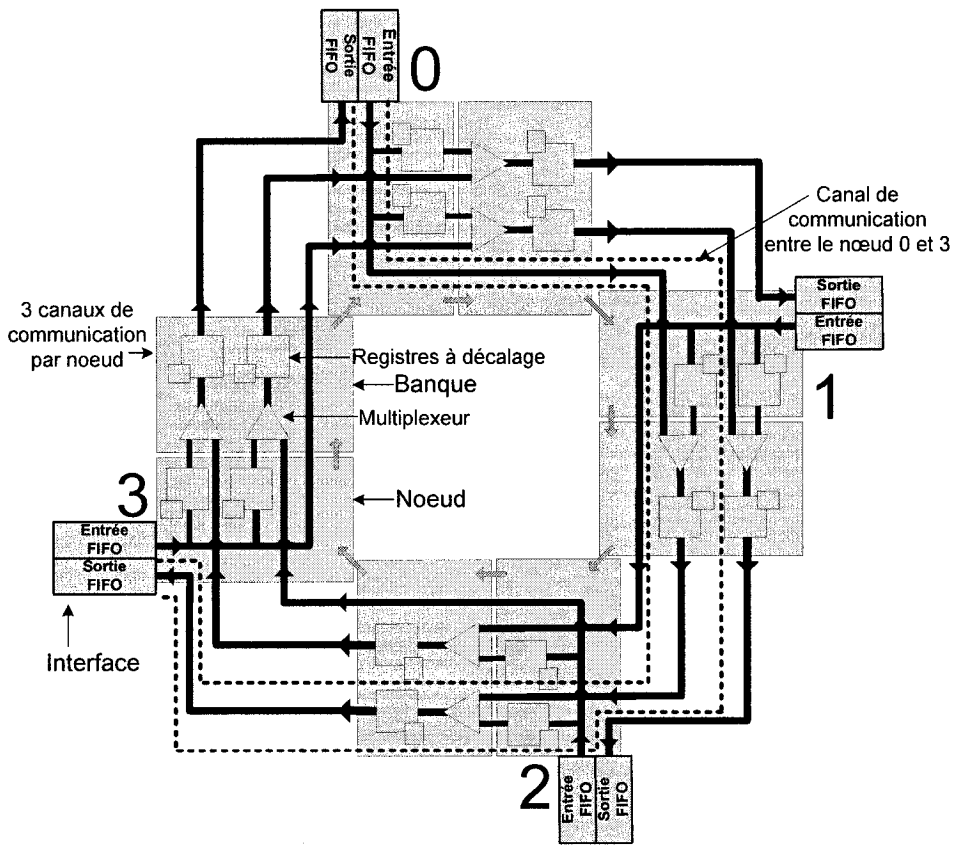


Figure 3-1. Modèle du RoC bas niveau implémenté selon les registres à décalage pour N = 4

3.1 Implémentation du canal de communication

Lorsque les paquets effectuent les sauts d'un nœud à l'autre, ils se déplacent d'un tampon à l'autre à travers un canal de communication. Ce canal est simplement une chaîne de tampons de banque intercalée de multiplexeurs. L'exemple d'un canal est présenté à la **Figure 3-1**. Il est délimité par les lignes pointillées et celui-ci permet la communication du nœud 0 au nœud 3. Il peut également servir à la transmission du nœud 1 au nœud 3 et du nœud 2 au nœud 3.

Pour chaque nœud, n-1 canaux de communication existent et permettent d'atteindre n-1 destinations. De façon logique, un nœud ne peut envoyer un paquet à lui-même et ne peut pas recevoir un paquet de son voisin n-1. Ainsi, le canal situé le plus près du centre dans chaque nœud est toujours libre d'utilisation, car c'est à cet endroit que le chemin du canal débute. Cette partie du canal ne comporte aucun tampon et multiplexeur. Un paquet empruntant ce chemin est directement envoyé au prochain nœud et permet d'accélérer la transmission vers une destination éloignée. L'implémentation d'une telle structure obtient une complexité en ressource de $O(N^2)$.

3.2 Implémentation du Nœud

Les paquets sont composés de quatre champs : l'adresse source, l'adresse destination, les données et un champ d'information laissé libre pour l'utilisateur (*user bits*). Pour l'implémentation, la taille choisie des paquets est de 41 bits (Figure 3-2) : quatre bits pour le champ d'une adresse, 32 bits pour les données et un bit pour identifier si le paquet est pour une lecture ou une écriture.

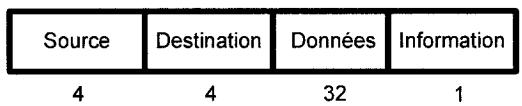


Figure 3-2. Format du paquet utilisé pour l'implémentation

Lorsqu'un paquet entre dans le réseau, il est transféré du FIFO d'entrée vers un tampon nommé FFD1 (*Flip-Flop D*) appartenant au premier niveau. Ce niveau 1 fait partie du nœud et permet de distinguer les étapes du déplacement d'un paquet (**Figure 3-3**). Pour un paquet donné, le choix du tampon dépend de son adresse de destination.

Une structure de type bitmap semblable à celle présenté à la section 2.5.1 est utilisée pour l'implémentation du modèle bas niveau. Son implémentation n'est pas totalement identique au modèle haut niveau, car en matériel la matrice des mémoires permet au nœud de transmettre un maximum de paquets vers la banque. Chacun des registres du bitmap indique si le tampon renferme un paquet ou s'il est vide. Le registre de bitmap est initialisé à la valeur binaire « 1 » lorsqu'il est temporisé d'un paquet et est réinitialisé à la valeur « 0 » lorsqu'un paquet quitte vers un prochain tampon. La machine à état des nœuds contrôle les transferts du FIFO d'entrée vers les tampons (FFD1) et inscrit les registres du bitmap. Un registre de bitmap initialisé empêche toute écriture à son tampon avant que le registre ne soit réinitialisé. Ce fonctionnement est le même que dans la situation où un paquet doit être transféré au prochain niveau (FFD1 vers FFD2). Tout tampon FFD1 sert de pré-charge aux canaux de communication. Ils permettent de maximiser l'utilisation des canaux dans les situations de trafic élevé en écrivant simultanément plusieurs paquets par cycle. La banque interroge les bitmaps du nœud et permet tout transfert au prochain cycle vers dans un tampon de banque libre. Finalement, lorsqu'un paquet arrive à la fin du canal de communication, il est transféré vers le FIFO de sortie pour que le paquet quitte le réseau par l'interface.

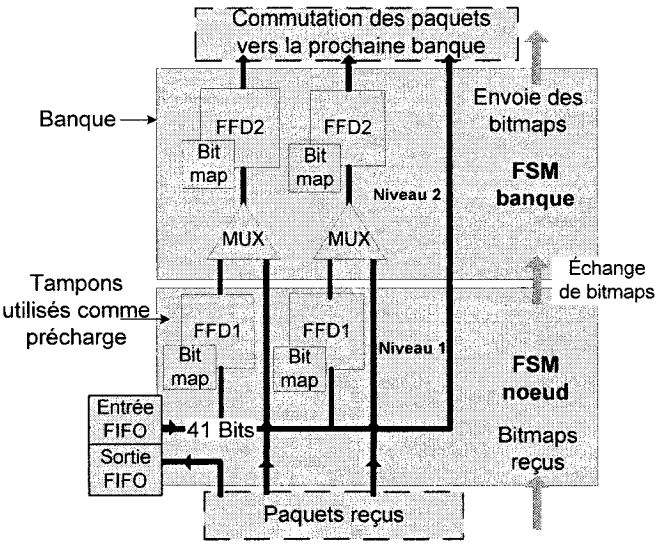


Figure 3-3. Modules du RoC avec ses chemins de données

3.3 Implémentation de la Banque

À chaque fois qu'un paquet se déplace vers une autre banque, il passe d'un tampon de banque près du centre vers une autre près de la périphérie. Comme dans le modèle de base du RoC, le saut d'une banque à l'autre se fait en sens horaire et à chaque cycle.

La seconde étape d'un paquet qui entre dans un nœud est de passé au tampon de banque du niveau 2 (**Figure 3-3**). Le transfert du tampon FFD1 vers le tampon de banque FFD2 se fait à travers un multiplexeur qui appartient à un canal de communication. La priorité étant réservée aux paquets déjà dans un canal de communication, le transfert s'effectue seulement si aucun paquet ne vient du tampon de banque précédent (voir le tracé pointillé de la **Figure 3-1**). En d'autres mots, le canal doit être libre d'utilisation pour qu'un paquet passe du nœud à la banque. La logique de contrôle autorise le transfert en utilisant l'information de bitmap de la banque précédente. L'utilisation des bitmaps permet ainsi aux machines à état et à la logique de contrôle de rapidement prévoir les chemins de données qui devront être sélectionnées à chaque multiplexeur.

3.4 Enveloppe de communication OPB

Pour que le RoC puisse communiquer avec des ressources utilisant des protocoles de communication différents, il est intéressant, pour sauver du temps de conception, que les interfaces du réseau s'adaptent sans modification majeure à l'architecture et à son fonctionnement. Pour cela, une enveloppe (en anglais, *wrapper*) basée sur le standard de communication en bus OPB (*On-chip Peripheral Bus*) [IBM03] a été développée pour connecter facilement le RoC avec ses ressources. D'autres protocoles auraient pu être choisis, mais le choix de l'OPB était lié à sa popularité d'utilisation par les blocs IP (*Intellectual Property*) et au protocole que supporte le processeur MicroBlaze [XILI00] de Xilinx devant être implémenté.

Conceptuellement, le modèle bas niveau du RoC a été conçu comme une série de couches qui englobe ses modules (chaque composante faisant partie du nœud et de la banque). L'enveloppe fait partie d'une de ces couches et elle se situe à la superficie des autres. Elle permet d'isoler la complexité des communications dans une simple machine à état et répond aux requêtes des ressources par ses interfaces maîtres et esclaves. Une partie de l'enveloppe se superpose à chacune des interfaces FIFO pour que les messages des ressources soient transmis aux nœuds comme des paquets à router.

3.5 Paramètres génériques

Inspiré du modèle de base du RoC et de son code en SystemC, le développement du modèle bas niveau pour une implémentation matérielle s'est fait en VHDL. Plusieurs itérations ont été nécessaires pour obtenir le modèle final. Tout d'abord réalisé avec seulement quatre nœuds pour connecter des ressources, le RoC est maintenant générique et permet d'ajuster différents paramètres.

Son architecture modulaire (appelé composante en VHDL) peut être aisément instanciée plusieurs fois et l'ajout de fonctionnalités est réalisable sans trop de modifications. Cette caractéristique est importante, car la modification du code pour une conception bas niveau requiert plus de temps que celui haut niveau. Les erreurs sont ainsi moins susceptibles de se produire et moins de temps sera nécessaire pour vérifier le fonctionnement du réseau durant la simulation. Tous les paramètres génériques présentés dans cette section renforcent l'aspect d'extensibilité du RoC.

3.5.1 Nombre de nœuds connectant les ressources au réseau

Le nombre de nœuds est flexible et son maximum pouvant être instancié dépend de la limite physique disponible au plan matériel. Ceci est le cas et sera vu dans les résultats. Au delà de 16 ou 20 nœuds, l'espace occupé par le RoC devient considérable et des alternatives à l'augmentation du nombre de nœuds doivent être envisagées. Peu importe le nombre de nœuds, l'architecture du RoC se reconfigure par elle-même. Le chemin de donnée (en anglais, *datapath*) est généré automatiquement et le réseau est ainsi facilement extensible. Cela permet à différentes explorations du nombre de ressources d'être analysées sans effort.

3.5.2 Adresse attribuée aux nœuds

Tout nœud a besoin d'une adresse l'identifiant comme une source ou une destination. Lors de la synthèse du RoC, chaque nœud obtient cette adresse ou plage d'adresse lui appartenant. L'adresse 1 est réservée au nœud 0 et les nœuds suivants obtiennent une adresse incrémentée de un à chaque fois. L'adresse 0 n'est pas utilisée, elle veut simplement dire qu'il n'y a aucun paquet. L'enveloppe qui entoure le RoC est générique et elle requiert quelques modifications pour générer des paquets en fonction des allocations d'adresses. L'enveloppe définie par une structure *package* VHDL s'adapte aux interfaces des FIFO pour répondre à des requêtes de maîtres ou d'esclaves des

ressources. L'aspect générique de l'adressage dépend aussi du nombre de ressources, car les premiers nœuds du RoC sont des maîtres et les suivants sont des esclaves.

3.5.3 Taille des champs à l'intérieur des paquets

Les quatre champs qui définissent un paquet peuvent être de n'importe quelle taille. Cela dépend des ressources matérielles disponibles. Plus un paquet est large, plus l'utilisation des ressources matérielles sera grande. Le champ d'adresse source et le champ d'adresse destination permettent au RoC d'être flexible sur le nombre d'adresses attribuées. Le champ *donnée* (**Figure 3-2**) permet d'adapter le débit que nécessitent les applications. Cet ajustement permet également d'obtenir une configuration respectant un ratio bande passante versus espace matériel. Finalement, le dernier champ du paquet, *user bits*, ajoute des identifiants pour intégrer d'autres fonctions au RoC (exemples : une priorité QoS, supporter d'autres protocoles, une adresse de niveau hiérarchique, un code de vérification d'erreurs CRC (*Cyclic Redundancy Check*)). Il est à noter que la largeur des paquets n'influence pas sur le fonctionnement du réseau, mais plutôt sur l'utilisation des ressources matérielles et possiblement même la fréquence maximale d'opération.

3.6 Génération du trafic

Comme tout design électronique, il est nécessaire de vérifier le bon fonctionnement. L'une des méthodes les plus simples pour tester le RoC et qui la rapproche d'une utilisation réelle est la génération de trafic pour stimuler le réseau. Plus précisément, un banc de test en VHDL simule les requêtes de communication OPB permettant de vérifier le réseau. Après les étapes de synthèse, d'assignation (de l'anglais *mapping*) et de placement & routage, ce même banc de test est réutilisé sur le RoC pour vérifier par simulation sur ordinateur (en anglais, *Personal Computer*, PC) si des paquets se sont perdus. Par contre, le VHDL n'est pas la solution parfaitement adaptée pour vérifier le fonctionnement du RoC puisqu'à ce niveau les modifications sont longues et fastidieuses.

De plus, il est difficile à ce niveau d'abstraction d'obtenir des analyses de performances du réseau.

3.6.1 Banc de test en langage *e*

Une autre méthodologie de banc de test peut être développée en langage *e*. Un tel banc de test établit les connexions avec les d'entrées et sorties du RoC vers le logiciel Specman [CADE00] qui exécute le code en langage *e*. Cette méthode est très intéressante, mais étant donné la quantité de travail à effectuer dans ce projet, la conception d'un banc de test en langage *e* se fera dans un travail ultérieur (section des travaux futurs).

3.6.2 Banc de test avec une application logiciel

L'utilisation de code logiciel en C/C++ permet de développer rapidement différentes applications pouvant tester le réseau. En exécutant le code sur les ressources, il est alors possible de générer plusieurs requêtes de communication au réseau. Le chapitre 4 (section 4.2) discutera de la partie du projet utilisant ce code logiciel pour tester et vérifier le RoC dans une architecture SoC. Ce code a été utilisé pour des simulations sur PC et pour une exécution sur FPGA. Différentes versions du code ont été nécessaires au fur et à mesure de l'intégration du RoC avec les ressources du SoC. Tout d'abord codé comme une simple demande d'écriture et de lecture au réseau, ce code a finalement permis d'exécuter une application roulant sur différentes ressources à la fois. Il a également permis de créer de multiples requêtes d'écriture et de lecture afin d'augmenter la bande passante du réseau.

3.7 Outils

Pour concevoir le RoC, différents outils sont nécessaires. Ceux-ci permettent d'effectuer la compilation et la synthèse du design VHDL. La plaquette de développement a servi

comme environnement de prototypage pour la vérification du réseau dans une puce FPGA. À partir du code VHDL pour le RoC et les fichiers de simulations fournis en annexe, la synthèse du RoC peut être refaite sans aucune erreur et les résultats présentés dans les prochains chapitres sont ainsi vérifiables. La version des outils, plus particulièrement pour la synthèse, a une incidence très importante. D'une version à l'autre, il se peut que de nouvelles erreurs apparaissent et rendent ainsi l'implémentation impossible. Voici la liste des outils utilisés pour réaliser cette synthèse :

1. Xilinx [XILI00] ISE 7.1.04i Project Navigator pour la compilation, la synthèse, le placement & routage, l'attribution des broches d'entrées et de sorties, ainsi que la génération du bitstream pour configurer la puce FPGA.

2. Synplify [SYNO00] Pro 7.5 et ou 8.4 de Synplicity pour obtenir de meilleurs résultats de synthèse. Cet outil permet de visualiser sur différents schémas l'architecture du réseau et ainsi déceler des erreurs de routage.

3. Xilinx EDK 7.1.02i Plateform Studio pour concevoir les ressources, qui connecté au RoC permettent de tester et vérifier ce dernier. L'outil se charge de générer les fichiers de simulation et de synthèse. Son interface est simple d'utilisation et permet de concevoir toute sorte de systèmes avec processeurs et autres blocs matériels. De plus, les applications en C qui s'exécutent sur les processeurs des ressources sont programmées et compilées avec cet utilitaire.

4. ModelSim [MENT00] SE 6.0d de Mentor Graphics pour la simulation pré et post-synthèse du RoC. Modelsim a permis également la simulation sur PC du SoC

roulant une application (le SoC est en fait le RoC incluant ses ressources connectées).

5. Carte de développement AP100 de Amirix [AMIR04] incorporant une puce FPGA Virtex-II Pro XC2VP100 de Xilinx [XILI00] avec une vitesse de grade (en anglais, *speedgrade*) -6 et un boîtier de forme FF1696.

CHAPITRE 4

INTEGRATION DU RoC A UNE ARCHITECTURE MULTIPROCESSEURS SUR PUCE

Pour vérifier le fonctionnement des réseaux intégré sur puce, il est possible de faire la validation par simulation à un haut niveau d'abstraction (modèle transactionnel) ou avec un plus bas niveau d'abstraction (e.g. niveau transfert de registres). Pour une réalisation sur circuit dédié (e.g. ASIC), il est aussi possible de faire une émulation sur FPGA. En résumé, plus la méthodologie utilisée implique une validation avec des détails proches de ceux de de la cible fixée, plus elle permet d'obtenir une analyse véritable du comportement du réseau. Dans notre cas, le choix de notre méthodologie est axé sur un FPGA. Nous utilisons le FPGA comme prototype d'un circuit dédié (e.g. ASIC), mais nous croyons également que le RoC pourrait être utilisé sur FPGA pour des solutions permanentes. Pour notre application le RoC sera connecté à plusieurs processeurs exécutant une application qui génère des requêtes pour le réseau.

Avant de faire l'implémentation (émulation) sur FPGA, toute partie d'un design se doit d'être simulée sur PC (e.g. niveau transfert de registres) pour éviter toutes pertes de temps que peut occasionner son déverminage. C'est pour cette raison qu'après chaque étape de synthèse, un banc de test en VHDL est appliqué au RoC. Lorsque la vérification est complétée, la prochaine étape consiste à l'intégrer et à le simuler avec le reste du système pour ainsi former un système multiprocesseurs sur puce (en anglais, *Multi-Processor System on Chip*, MPSoC). À la fin, ce MPSoC comprend le RoC et les ressources qui permettent de le vérifier sur FPGA en exécutant une application réelle.

4.1 Modèle bas niveau du système intégrant le RoC

Étant générique, le RoC peut connecter autant de ressources (e.g. processeurs) que voulu. Le modèle présenté à la **Figure 4-1** a été développé pour tester le RoC dans une application SoC. Le modèle est basé sur une architecture à mémoire distribuée. Chaque ressource exécute une tâche qui permet de générer des requêtes de communication pour le réseau. Pour gérer l'application, la ressource 0 est considérée comme maître. Elle partage sa seconde mémoire avec les autres ressources « esclaves » 1 à N pour échanger des messages. Dans cette mémoire, résident les tâches créées par le processeur 0. Les ressources esclaves accèdent par le réseau à la mémoire partagé pour récupérer les tâches et écrire à la mémoire les résultats des tâches exécutées. Le processeur 0 accède de temps à autre à sa mémoire partagée pour récupérer les résultats et écrire de nouvelles tâches à exécuter par les ressources esclaves. Les ressources sont considérées esclaves, car elles ne font qu'effectuer les tâches demandées par le processeur 0, maître de l'application.

Par l'entremise de communications sérielles, réalisées avec l'aide d'un UART (*Universal Asynchronous Receiver Transmitter*) disponible à l'intérieur de la ressource 0, l'application transmet des informations du système vers un PC. Ainsi, lorsque le SoC est implémenté sur FPGA, les informations reçues permettront de vérifier le fonctionnement de modules du SoC, de l'application exécutée et bien évidemment du RoC. Dépendamment de l'application, elle peut identifier quelle partie du réseau est défectueuse.

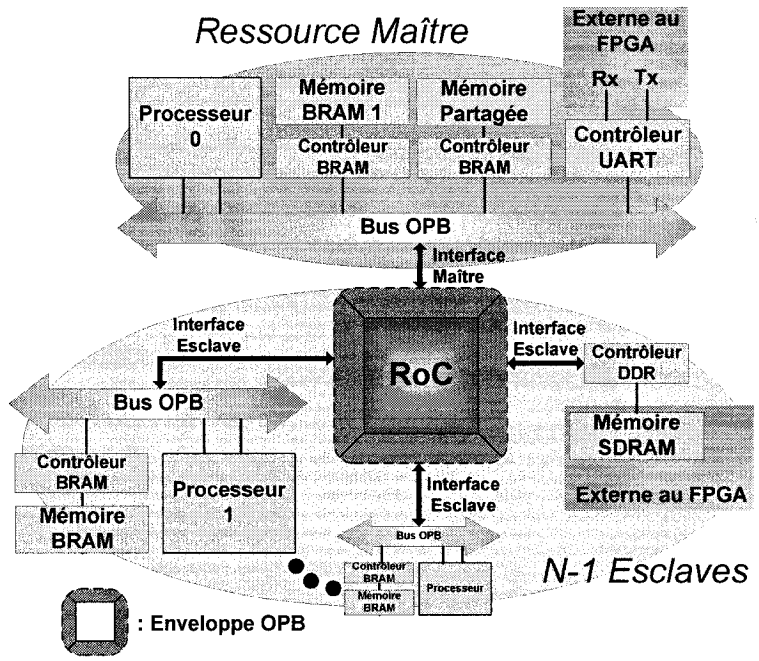


Figure 4-1. Modèle bas niveau du SoC incorporant le RoC

Les mémoires utilisés par les processeurs sont de type BRAM (*Block Random Access Memory*). Ces mémoires BRAM disponibles sur un FPGA sont limités. Ainsi, pour permettre d'exécuter une application plus exigeante en espace ou nécessitant d'accéder à des informations trop nombreuses pour résider sur les mémoires BRAM, une ou plusieurs mémoires SDRAM (*Synchronous Dynamic Random Access Memory*) externes au FPGA peuvent être utilisées sur la carte de développement. Pour utiliser une mémoire externe, aucune modification à l'architecture du RoC n'est nécessaire. Il suffit d'utiliser un contrôleur de mémoire DDR (*Double-Data-Rate*) et le connecter à une interface esclave du RoC. L'utilisation d'une mémoire externe est une fonctionnalité intéressante, mais elle n'est pas présentement utilisée.

À l'intérieur de chaque ressource se trouve un processeur MicroBlaze, une mémoire BRAM et des périphériques (contrôleur de mémoire et UART) connectés par un bus de type OPB pour les communications locales. Chaque bus est connecté au RoC par les interfaces OPB de son enveloppe. Dépendamment de l'interface, la ressource est considérée comme maître ou esclave. L'interface esclave permet à un processeur de faire une requête via le RoC comme si ce dernier est une composante esclave de la ressource. Pour que le RoC transmette une requête d'écriture ou de lecture vers la mémoire partagée, il doit ensuite être connecté par une interface de type maître pour accéder au bus OPB (voir Figure 4.1).

L'architecture du modèle est simple et rapide à concevoir et différentes alternatives sont envisageables pour améliorer le système. Pour désengorger une interface maître que les esclaves sollicitent fréquemment pour accéder à la mémoire partagée, l'application et les interfaces peuvent être modifiée de façon à ce que le maître communique à des mémoires partagées à l'intérieur de chaque ressource esclave. Par contre, il nécessite plus de ressource mémoire en BRAM du FPGA et il ralentit le fonctionnement du processeur 0, car il doit constamment communiquer à travers le réseau. Bref, le choix de la configuration d'un MPSoC est essentiellement influencé par les besoins de l'application et des ressources matérielles disponibles.

Une autre configuration du système aurait été d'utiliser les liens de communication point à point unidirectionnel (en anglais, *Fast Simplex Link,* FSL). Ceux-ci sont simples et plus rapides que le bus OPB. Par l'entremise de FIFOs, les données sont échangées entre deux composantes en seulement deux coups d'horloge. Dans le cas d'une application nécessitant des communications fréquente avec sa mémoire partagée, utiliser les FSL est un avantage intéressant. Malgré ce fait, nous sommes restés avec notre choix de l'OPB. Tout d'abord, il est un standard compatible avec plusieurs composantes IP. De plus, si le

RoC devait utiliser les liens FSL, il faudrait une interface nœud pour chacune des composantes à l'intérieur de la ressource.

4.2 L'application logicielle

Deux versions de l'application permettant de vérifier le RoC ont été envisagées. Chacune d'elle se sert de l'UART pour obtenir des informations sur le fonctionnement du système et du réseau. La première a été utilisée pour vérifier les simulations et l'implémentation. La seconde ne fait uniquement que l'objet d'une discussion.

4.2.1 Application no 1 : Écriture et lecture multiples

La première application développée effectue plusieurs lectures et écritures entre chaque processeur et la mémoire partagée. Le processeur 0 vérifie si chacune des opérations s'est produite correctement. Cette méthode s'est avérée suffisante pour vérifier le fonctionnement et identifier quelle partie du réseau ou des ressources est défectueuse pour les simulations sur ordinateur ou pour l'implémentation sur FPGA.

Par contre, pour convenablement vérifier le RoC et identifier les limites de son fonctionnement (en remarquant la modification ou la perte d'un paquet dans le réseau), une deuxième application avec un trafic plus important est nécessaire.

4.2.2 Application no 2 : algorithme de tri

La nouvelle version de l'application est une suite logique des améliorations nécessaire pour évaluer le RoC. Cette application n'a pas été réalisée, mais dans le reste de cette section nous présentons un résumé.

Idéalement, l'application devrait rester modeste pour qu'elle puisse résider dans les mémoires BRAM et que le modèle soit capable de l'exécuter. Elle doit effectuer de vraies opérations pour répliquer certains comportements d'une application réelle. Une application exécutée en parallèle sur des processeurs obtient certaines granularités. La granularité d'une application est le rapport entre le temps passé à effectuer les calculs et le temps passé dans les communications). Cette granularité peut être différente d'un processeur à l'autre qui exécute l'application, car les temps de communication dépendent du réseau. Il est donc intéressant d'observer le comportement du réseau en fonction de sa configuration et de l'application exécutée. Ces observations permettent d'effectuer des ajustements à la configuration du réseau ainsi qu'au code de l'application. Que ce soit des opérations mathématiques, des traitements d'images ou des manipulations de données, l'application génère des requêtes d'accès au réseau et permet ainsi d'obtenir une évaluation de la performance du système. Pour vérifier le fonctionnement du réseau avec cette application, celle-ci est simulée préalablement sur ordinateur pour connaître les résultats que devait obtenir l'implémentation. Ainsi, après l'exécution, l'utilisateur est informé des opérations effectuées sans erreurs (paquets arrivés à destination avec les bonnes données) obtenant ainsi des indices sur les parties défectueuses.

Afin d'établir de nombreuses requêtes de communication, une des applications envisagées est l'utilisation d'un algorithme de tri. Par exemple, le tri par fusion requiert peu de ligne de code pour le concevoir et est exécutable sur n'importe quel processeur. Il est ainsi intégrable au modèle de base. Le tri s'effectue sur chaque processeur esclave qui récupère les listes de données à ordonner du processeur maître. En général, pour ordonner un ensemble de données, celles-ci sont divisées en deux parties égales. Chaque partie est divisée en sous parties et le tri s'effectue sur au moins deux échantillons compris dans une sous partie. Lorsque les sous partis sont correctement ordonnés, elles sont fusionnées en un ensemble. Ainsi de suite, l'ensemble mieux ordonné est divisé de nouveau en partie jusqu'à ce que le tri soit complété.

4.3 Méthodologie de conception pour multiprocesseurs sur puce

Les outils ISE et EDK de Xilinx ont apporté un gain important dans la conception de notre système. Ces outils offrent gratuitement des composantes IP qui peuvent être utilisées autant de fois que voulu (processeur MicroBlaze, contrôleur de mémoire BRAM, contrôleur de communication sériel, arbitre et bus de communication OPB). Plusieurs approches existent avec ces outils afin de joindre le RoC aux ressources du système.

On peut d'abord créer une seule ressource esclave dans l'outil EDK et la répliquer par la suite comme un projet dans l'outil ISE. Ainsi, cette ressource peut être réutilisée autant de fois que voulu et permet de sauver sur le temps de conception. De plus, cette méthode peut permettre d'éviter des erreurs de conception pour chaque nouvelle ressource. Malgré le temps investi pour faire fonctionner cette méthode, nous nous sommes rendu compte que les outils ne sont pas encore adaptés pour ce type de conception envisagé. De nombreuses erreurs de synthèse et un problème de gestion de la chaîne de *scan* associé à chaque ressource qui est dupliqué empêchent l'utilisation de cette méthodologie. Peut-être qu'avec les versions futures de ISE, l'outil supportera plus d'un projet EDK à la fois.

L'approche que nous avons finalement adoptée est légèrement différente. Le RoC est d'abord développé dans ISE avant de le connecter aux ressources du système. L'ensemble des ressources d'un projet EDK est exporté vers ISE avec l'option « export to project Nav » de EDK. Une attention particulière est apportée à la configuration du projet EDK pour la création des broches d'entrées et de sorties du bus OPB qui doivent se connecter à l'enveloppe (en annexe sont fournis les fichiers VHDL pour la création des connexions OPB maître et esclave).

Lorsque le projet EDK est réuni au RoC dans l'outil ISE, un fichier VHDL de haut niveau (*top level*) permet de joindre les connexions entre l'enveloppe du RoC et les interfaces OPB des ressources. Le fichier des contraintes UCF (*User Contraint File*) et le fichier d'initialisation des mémoires BRAM doivent être spécifiés pour que le projet ISE soit prêt pour une synthèse complète du système. Lorsque le fichier binaire (bitstream) est obtenu, une dernière manœuvre est nécessaire pour que le système fonctionne correctement. Le bitstream doit être importé de nouveau vers le projet EDK pour que les mémoires BRAM soient correctement initialisées et que l'application puisse être exécutée. EDK génère ainsi un nouveau bitstream pour configurer le FPGA. Cela permet d'effectuer des itérations au logiciel des processeurs sans devoir refaire la synthèse. Si aucune modification matérielle n'est appliquée au système, le logiciel est compilé et réinséré dans les BRAM lors des réinitialisations. On peut ainsi rapidement obtenir une configuration du FPGA servant uniquement à identifier des parties défaillantes du système. D'un autre côté, si une partie matérielle est modifiée (que ce soit l'adressage, les connexions ou le fonctionnement d'un module), la synthèse des projets EDK et ISE doivent être refaits.

4.4 Intégration du RoC avec un tunnel HyperTransport

Pour repousser les limites des performances des systèmes et permettre d'évoluer vers de nouvelles applications, de nouvelles méthodes doivent être introduites à la conception. Interconnecter plusieurs puces, chacune possédant un MPSoC, est une méthode pouvant être conçue rapidement et prometteuse en performances. L'application n'est plus limitée à une seule puce, mais est extensible selon les puces ajoutées.

Peu d'ouvrages dans la littérature aborde le sujet des moyens de communication permettant de joindre plusieurs systèmes entre eux. Afin d'apporter une nouvelle dimension au projet du RoC, nous avons exploré son intégration à une chaîne de communication interpuce via le tunnel HyperTransport.

La section suivante se penche sur la notion de pont ainsi que sur un modèle qui a permis de joindre le RoC et le tunnel HyperTransport (HT). Ce travail a été réalisé en collaboration avec M. Ami Castonguay [CAST06], étudiant à la maîtrise, qui a travaillé à la synthèse de ce tunnel. En particulier, nous présentons brièvement le pont et le tunnel HT, mais pour plus de détails le lecteur doit se référer au mémoire de M. Castonguay. Étant donné que le projet de recherche se concentre d'abord sur la conception du RoC, nos contributions se concentrent ici davantage à la définition des requis du pont, à l'interconnecter avec le RoC et le tunnel HT, à simuler les modules et à les déverminer par la suite pour qu'ils fonctionnent en harmonie. Par conséquent, le modèle développé permet de valider l'intégration du RoC avec le tunnel HT, toutefois plusieurs caractéristiques devraient être ajoutées pour une utilisation courante avec de vraies applications.

4.4.1 Le tunnel HyperTransport

Tout d'abord, l'HyperTransport (HT) est un protocole de communication interpuces très performant qui interconnecte une grande variété de composantes. Sa topologie de base est une chaîne de composantes connectées point à point qui peut également offrir une topologie en étoile grâce à des commutateurs HT. Une chaîne est formée de plusieurs éléments composés principalement d'un hôte, d'un élément de fin de chaîne nommé *cave* ainsi qu'un nombre variable de tunnels. Le cave est un élément de fin de chaîne, ce n'est pas une destination, mais un élément de contrôle. Chaque tunnel est formé de deux liens de communication HT tel qu'illustré à la **Figure 4-2**. Les communications à travers le HT sont routées par l'hôte de la chaîne. Un mode de communication appelé *DirectRoute* permet aux paquets de passer directement par les deux parties du tunnel HT. De cette façon, les paquets ne transitent plus vers l'hôte, ce qui permet ainsi de réduire la latence des communications dans le tunnel.

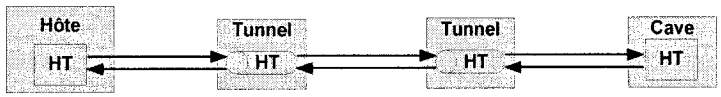


Figure 4-2. Topologie d'une chaîne HT

La chaîne HT a un système d'adressage de 40 ou 64 bits qui lui permet d'acheminer les messages. Sa méthode de transport est semblable à celle utilisée dans les NoC, c'est-à-dire une méthode basée sur la temporisation et retransmission (en anglais, *store and forward*). Pour cela, chaque nœud HT comporte trois tampons indépendants pour supporter les trois types de messages à transmettre :

- Posted – requête simple. Comme une écriture, elle ne requiert aucune réponse.
- Non-Posted – requête qui doit recevoir une réponse, comme dans le cas d'une lecture.
- Response – requête permettant de répondre à un message Non-Posted.

Les messages sont composés d'un ou plusieurs paquets. Un paquet d'en-têtes, nommé paquet de commandes, est le premier paquet du message et sert d'informateur sur le type de paquet transmis, de sa destination et du rôle des données servant pour les opérations. Si le type de message le requiert, un paquet de données est associé au paquet de commande et comprend de 4 à 64 octets.

Il existe également les paquets de types NOP qui servent au contrôle du flux des communications. Ceux-ci n'ont pas besoin d'être emmagasinés à l'intérieur de tampons. Ils sont transmis lorsque aucun message (ou paquet) n'a besoin d'être transmis. C'est à l'intérieur d'un de ses champs qu'est incorporée l'information sur l'état des tampons.

4.4.2 Spécifications du pont

Un pont est une interface physique qui inclut un protocole et qui permet de lier deux bus ou encore un bus et un NoC (RoC). Ici un pont servira à lier le RoC et le tunnel HT. Comme ces derniers échangent des messages par l'entremise d'une mémoire partagée, ceci rend la tâche de conception plus simple pour permettre les échanges de paquets. Les paquets sont adaptés lors de leur transmission du RoC vers la chaîne HT et de la même façon, les requêtes provenant de la chaîne HT sont adaptées en paquets pour le RoC.

Le pont HT permet d'adapter les modes de lecture sensiblement différents du HT et du RoC. Pour ce faire, il doit garder en mémoire des informations sur les paquets de types lecture transmis. Cette situation est le cas lorsqu'un message de lecture provient de la chaîne HT et atteint le RoC. Pour générer une réponse au HT, le pont doit avoir l'adresse source du message. De la même manière, si le RoC génère une requête de lecture vers le HT, le pont doit garder en mémoire la provenance du message pour envoyer les données à la bonne ressource.

Cette adaptation nécessite un ajout au pont. Il doit assurer l'échange de communications du côté HT et RoC sans influencer leur fonctionnement. Dans une situation où les espaces mémoires sont entièrement utilisés, le tunnel refuse les opérations et attend que des réponses soient reçues avant de libérer les mémoires. Une situation d'interblocage peut se produire dans le cas où deux communications s'attendent mutuellement. Le pont doit s'assurer qu'une requête de lecture ne peut pas bloquer le passage d'une réponse, car même si le protocole HT prévoit une telle situation, certains systèmes ne la prévoient pas. Pour y remédier, deux ports de communication du RoC sont utilisés par le pont HT. Le premier sert à traiter les requêtes et le second est dédié aux réponses. En utilisant deux ports indépendants, cela permet d'isoler les types de transmissions qui peuvent s'inter bloquer.

Également, comme c'est le cas de plusieurs bus, le pont supporte l'envoi et la réception en rafale pour le transfert de grande quantité de données. Cette transmission permet de réduire le nombre d'opérations nécessaire pour l'envoi d'une quantité importante d'information. Évidemment, l'HT supporte ce moyen de communication. Normalement, chaque paquet a son propre en-tête, si le même en-tête est utilisé pour l'envoi de plusieurs paquets, cela réduit le trafic à l'intérieur de la chaîne HT et augmente la bande passante. Cette communication en rafale a par contre le désavantage d'accroître la latence, car le pont accumule les données jusqu'à un maximum de 64 octets avant d'envoyer un paquet HT.

4.4.3 Détails de fonctionnement du pont

À la base, le pont est en fait un simple traducteur de paquets. Deux modules principaux servent pour la traduction : le premier module convertit les paquets du tunnel HT vers le RoC et le second réalise le chemin inverse. Pour rediriger les réponses au bon destinateur, le pont contient deux tables de conversion pour mémoriser la provenance des demandes de lectures. Une table sert à mémoriser ceux en provenance du tunnel et l'autre en provenance du RoC.

Pour prendre avantage du mode de communication en rafale du tunnel HT, une mémoire embarquée est ajoutée à l'architecture du pont et permet d'emmagasiner les paquets avant de les envoyer en masse. Présentement, le RoC ne supporte pas les transmissions en rafale, mais les versions futures pourront intégrer ce mode de communication. De plus, ce mode permet d'ajuster facilement le pont aux différentes tailles des paquets du RoC.

4.4.4 Modèle pour valider l'intégration

Pour fonctionner correctement, la chaîne HT nécessite absolument un élément HT de type hôte pour initialiser chacun des éléments et assigner les adresses aux tunnels HT. Cet élément est long à concevoir et il était manquant au début de l'intégration. Pour cette raison, l'hôte HT a été conçu en logiciel à un niveau transactionnel. Ce niveau limite l'intégration au niveau de la simulation, car pour une implémentation il faudrait synthétiser cet élément. En plus, l'espace disponible sur la puce FPGA est limité et ne peut pas permettre l'implémentation du modèle au complet.

L'architecture du modèle présenté à la **Figure 4-3** est composée de deux blocs quasiment identiques, l'adressage et le logiciel changent légèrement. Pour accélérer la conception, deux instances d'un MPSoC du RoC à quatre ressources sont utilisées. Pour cela, deux ressources du MPSoC sont substituées pour connecter le RoC aux deux ports du pont. L'enveloppe est également substituée des connexions deux et trois pour que le pont se connecte directement aux FIFOs du RoC. L'ensemble du système fonctionne sur le même signal d'horloge et de réinitialisation. Ces signaux sont générés par le tunnel HT.

Le logiciel est semblable à celui mentionné à la section 4.2.1 Au lieu de tester les accès vers les mémoires partagées du MPSoC local, les accès se font vers les mémoires partagées de l'autre MPSoC. Pour atteindre le pont, le paquet doit être routé vers le bon port du RoC. Le pont traduit le paquet en format RoC en paquet format HT et ce dernier est transmis au prochain tunnel HT. De l'autre côté, un autre pont récupère ce paquet en format HT et le traduit en paquet format RoC.

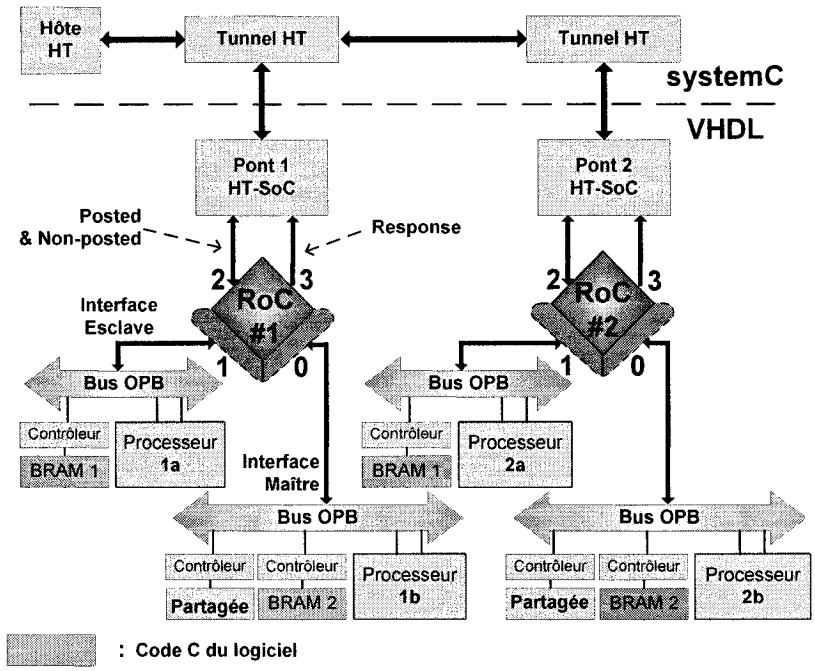


Figure 4-3. Modèle intégrant les systèmes MPSoC du RoC avec la chaîne HT

Le logiciel génère toutes les sortes d'accès possible à la mémoire partagé pour vérifier les chemins du système. Pour que le logiciel envoie un message d'un MPSoC vers l'autre MPSoC, l'adresse choisie détermine à quel nœud le paquet est envoyé dans le RoC. Lorsque le paquet atteint le pont, la configuration du pont entraîne automatiquement un routage vers l'autre MPSoC. Notez que plus qu'un RoC peut être connecté à l'ensemble du système avec les tunnels HT. Ainsi, les ponts ont seulement un RoC comme destination. Ceci simplifie la conception, mais limite comme mentionné l'aspect générique du design. Si plus de deux RoCs devaient être connectés par un tunnel, il serait alors possible (via l'enveloppe OPB) d'insérer dans le champ *user bits* d'un paquet, le numéro du RoC comme destination. Ainsi, l'enveloppe décoderait l'adresse d'une requête devant atteindre un autre RoC que celui local.

L'ensemble du modèle est simulé avec Modelsim. Celui-ci compile et réalise la simulation de chaque langage de conception faisant parti du modèle (voir l'identification

de chaque langage sur la **Figure 4-3**). Le pont est le premier élément simulé en VHDL, ensuite après avoir intégré l'ensemble du modèle, une simulation de l'application logicielle est exécutée sur chaque processeur pour effectuer des tests d'écriture et lectures.

Bien qu'une validation par simulation soit réalisable, le modèle complet n'est pas parfaitement synthétisable. En effet, l'Hôte étant modélisé au niveau transactionnel, il n'existe pas pour le moment d'équivalent précis au niveau cycle (en anglais, *cycle accurate*). Pour le reste, l'ensemble des éléments VHDL est synthétisable avec les outils de Xilinx et le tunnel est synthétisable avec Synplify.

CHAPITRE 5

RESULTATS ET ANALYSE

Pour évaluer les performances du RoC, plusieurs synthèses ont été générées en modifiant chacun des paramètres du réseau (section 3.5). Les résultats concernent deux principaux paramètres dont celui de l'utilisation des ressources matérielles et celui des capacités de communication. Ces paramètres influencent grandement les résultats et nous permettent de choisir les configurations du réseau en fonction des besoins en performance (e.g. latence et bande passante). Nous allons donc d'abord explorer les différents compromis coût/performance.

Les résultats et les analyses de l'intégration du RoC au tunnel HyperTransport font également partie de ce chapitre. Finalement, après avoir vérifié le RoC par simulation sur ordinateur et l'avoir implémenté sur FPGA, nous avons comparé le RoC à d'autres réseaux intégrés sur puce. Cette comparaison met en évidence les points forts et les points faibles du RoC. Notez que chaque résultat de synthèse dans ce chapitre utilise un format de paquets de 41 bits (32 bits de données), excepté lors de la comparaison du RoC avec des paquets de tailles différentes (section 5.1.3).

5.1 Analyse de performance du RoC

Les résultats de synthèse présentés au **Tableau 5-1** concernent uniquement le RoC. Aucune ressource (processeur, mémoire ou contrôleur) n'est connectée au réseau. L'enveloppe OPB fait partie de la synthèse et elle ne représente qu'une faible partie occupée de la complexité en tranches (en anglais, *slices*) et en bascule (en anglais, *Flip Flops* qu'on abrège par FF). Les *tranches* sont des composantes à l'intérieur du FPGA et permettent de créer la logique combinatoire. Les FFs sont les éléments de mémoire à l'intérieur des tranches et sont synchronisé avec l'horloge (deux FF sont présents par

tranche). Lorsque l'on veut interpréter l'espace occupé par un design, le nombre de tranches et de FF sert de référence.

Les résultats de latence et de bande passante sont obtenus avec une simulation utilisant un trafic peu élevé et des communications non bloquantes. Ces caractéristiques de la simulation permettent de faciliter l'obtention des résultats. Les mémoires des nœuds et des banques du RoC utilisent les mémoires BRAM distribués du FPGA pour réduire l'utilisation des ressources matérielles. Des résultats de synthèses plus détaillés sont disponibles en annexe.

Tableau 5-1. Résultats de différentes synthèses du RoC obtenues avec Synplify

Nombre de noeuds	Tranches	FF	Fréquence (MHz)	Latence (cycles)	Bande passante à un noeud (Mo/s)	Bande passante du RoC (Go/s)
4	682	1031	134	18	536	2,14
8	2628	4447	110	22	440	3,52
12	5835	10231	101	26	404	4,85
16	9754	17308	92	30	368	5,89

La synthèse d'un RoC à quatre nœuds utilise environ 171 tranches (682/4) pour chaque nœud avec sa banque associée. Les versions du RoC à 8, 12 et 16 nœuds consomment à peu de choses près 328 (2628/8), 487 (5835/12) et 610 (9754/16) tranches. Cela confirme que la complexité du RoC progresse selon $O(N^2)$ où N représente le nombre de nœuds (section 3.2). Cette complexité est également observable par l'augmentation du nombre de FF.

Comme tout circuit synchrone, le RoC a un chemin critique qui détermine sa fréquence d'opération maximale. Ce chemin est délimité par la portion de logique combinatoire

entre deux éléments de mémoire. Le temps pour parcourir ce chemin détermine la période minimale de l'horloge pour synchroniser le circuit. Les résultats de la synthèse indiquent que le chemin critique du RoC se situe dans le décodage de la machine à état. Durant cette étape, un paquet est transmis du FIFO vers un tampon de nœud. C'est par l'entremise des bitmaps et de l'adresse de destination du paquet que le décodage informe à chaque coup d'horloge si le tampon de nœud destiné est libre (voir la **Figure 3-3** pour visualiser le déplacement des paquets). À titre de travaux futurs, nous discuterons d'une approche pour réduire le chemin critique de cette machine à état et ainsi augmenter les performances du RoC.

5.1.1 Latence

La latence correspond au nombre de cycles d'horloge nécessaire pour qu'un paquet effectue son chemin à travers le RoC. Dans le cas d'une lecture, un paquet (requête) traverse le réseau jusqu'à la destination alors qu'un second paquet (donnée) revient à la source avec les données. Peu importe la configuration du RoC, l'équation 1 permet d'obtenir le nombre de cycles nécessaires pour effectuer une requête de lecture (où N est le nombre de nœuds).

$$L= 2 \times (8 + N - (N/2 +1)) \quad (1)$$

Dans le cas d'une écriture, un paquet (requête et donnée) traverse le réseau jusqu'à la destination alors qu'aucun paquet ne revient à la source. Notez ici q'un paquet (confirmation) aurait pu revenir à la source, mais pour des raisons d'optimisation nous l'avons omis. Le nombre de cycles nécessaire pour une écriture (sans devoir renvoyer un paquet de confirmation) est présenté par l'équation 2. Dans cette équation, N représente le $n^{ième}$ nœud qui reçoit le paquet à partir du nœud source. Par exemple, la destination la plus grande (N-1) d'un RoC à 6 nœuds a une latence de 13 cycles.

$$L= 8 + N \quad (2)$$

5.1.2 Bande passante

En plus des résultats qui concernent la latence, la bande passante est l'un des critères les plus importants qui mettent en valeur les performances du RoC. Selon les résultats à la **Figure 5-1**, nous remarquons que le réseau permet de transmettre d'importante quantité d'information. Ces résultats sont basés sur la fréquence d'opération générée par l'outil de synthèse. Pour chaque configuration du RoC (4, 8, 12, 16 nœuds, etc.), le paquet transmet des données de 32 bits. Plus le nombre de nœuds augmente, plus la quantité de matériel augmente. Par conséquent l'outil de synthèse éprouve de plus en plus de difficulté à minimiser la latence et donc l'horloge du système. Les synthèses obtenues affichent des fréquences d'opérations réduites, ce qui a pour conséquence de diminuer la bande passante aux nœuds. La bande passante agglomérée augmente de façon linéaire, car chaque nœud augmente le débit du réseau à coup de 400 Mo/s (*Mégaoctet par seconde*). Toutefois, on observe (**Figure 5-1**) que cette bande passante par nœud reste constante (400 Mo/s) au fur et à mesure que le nombre de nœuds augmente. Par exemple, un RoC à 16 nœuds fonctionnant à 92 MHz supporte une bande passante de 6 Go/s et ce dernier est distribué uniformément sur les 16 nœuds.

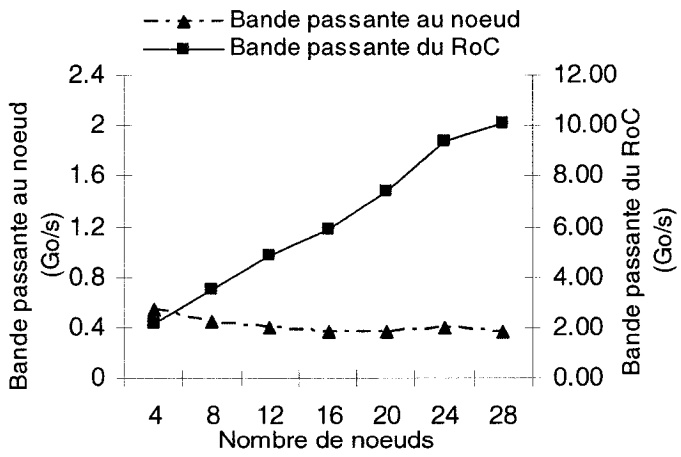


Figure 5-1. Bande passante à une interface et agglomérée du RoC

5.1.3 Ressources matérielles

Le RoC peut aussi supporter des paquets de taille différente. En modifiant sa configuration, les synthèses du RoC démontrent que sa complexité évolue selon O(N). La **Figure 5-2** rapporte l'espace occupé en pourcentage et en tranches. Lorsque la taille des paquets double, l'occupation du RoC sur le FPGA double également. La bande passante aux nœuds double aussi, puisque la synthèse donne les mêmes résultats de fréquence d'opération. Cette analyse est particulièrement intéressante dans le cas d'une application qui nécessiterait une plus grande bande passante. Elle démontre le coût relié à l'augmentation instantanée du débit souhaité pour le réseau.

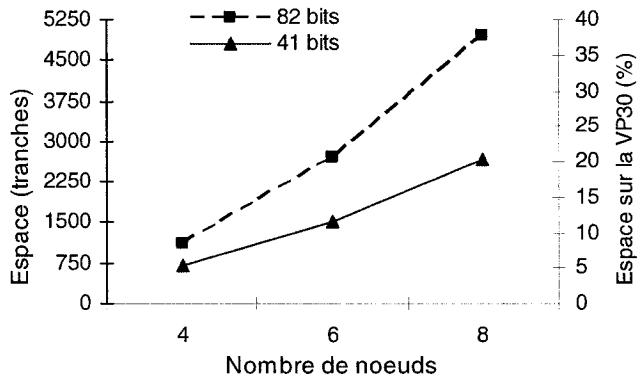


Figure 5-2. Augmentation des coûts lorsque l'on double la taille des paquets du RoC

Le fait que la fréquence maximale de synthèse reste la même lorsque l'on augmente la taille des paquets permet de jouer sur les coûts du RoC. Par exemple, si on considère qu'un RoC à 16 nœuds occupe trop d'espace et qu'une partie de sa bande passante peut être réduite, en diminuant la taille des données de 32 bits à 16 bits, le format des paquets passera de 41 bits à 25 bits. Cela représente une réduction de 40% en ressource matérielle. Ce RoC à 16 nœuds passera donc de 9754 à 5852 tranches, ce qui est l'équivalent en coût déjà celui d'un RoC à 12 nœuds. Finalement, la bande passante aux nœuds sera réduite par deux et passera de 368 à 184 Mo/s. Même chose pour la bande passante totale du RoC qui passe de 5,89 à 2,95 Go/s.

5.2 Analyse de performance du MPSoC

C'est par la synthèse du MPSoC que nous avons pu obtenir une analyse de complexité matérielle de l'architecture du RoC face à un système complexe.

5.2.1 Ressources matérielles du MPSoc

Tout d'abord, l'espace utilisé par le RoC augmente rapidement, mais son rapport avec l'ensemble du système demeure acceptable. Toutefois, plus le nombre de ressources (nœuds) augmente, plus le rapport d'occupation de l'espace du RoC face aux ressources augmente. À un point tel qu'il est préférable d'envisager des alternatives à l'architecture pour réduire sa complexité. Pour simplifier cette explication, le **Tableau 5-2** rapporte le nombre de tranches requis pour différents MPSoCs. Au total, ces MPSoCs incluent un RoC avec son enveloppe OPB et les ressources (processeur MicroBlaze, mémoire BRAM et autres périphériques). Une colonne de pourcentage indique l'utilisation de tranches par rapport au nombre maximal disponible sur une puce FPGA XC2VP100 [XILI00]. Deux autres colonnes informent du nombre total de BRAM nécessaire pour ce MPSoC. Notez que la VP100 possède un maximum de 44,096 tranches et 444 BRAM. Pour obtenir les résultats du tableau, nous avons extrapolé les résultats en tranches et en BRAM du MPSoC à 4 ressources et pris en compte les différents résultats de synthèse du RoC. Une seule ressource utilise 800 tranches. Pour obtenir l'occupation d'espace d'un MPSoC, il suffit d'utiliser l'équation 3. N est le nombre de ressources.

$$\text{Nombre de Tranches} = N \times 800 + (\text{RoC à N nœuds}) \qquad (3)$$

Le RoC occupe 43% d'un MPSoC à 16 ressources et 53% d'un MPSoC à 24 ressources. Ainsi, réduire la complexité du RoC peut s'avérer important lorsque l'on veut implémenter un tel MPSoC. Un design qui occupe 80% des tranches du FPGA et qui respecte les contraintes de temps peut-être difficile à générer par un outil de synthèse. La synthèse d'un MPSoC avec plus que 20 ressources n'est donc pas idéale. La réalisation

d'un MPSoC de taille importante ne peut être réalisé sans l'optimisation du RoC. Pour ce faire, il est préférable d'envisager des optimisations comme la hiérarchisation et la moitié des tampons pour diminuer l'espace occupé le RoC (voir section 2.5). Si on se fie à la loi de Moore, ce problème de ressources maximales pour plus que 20 nœuds sur une puce VP100 devrait être repoussé d'ici peu avec l'arriver des prochains produits de Xilinx (e.g. famille du Virtex-4).

Tableau 5-2. Résultats de synthèse pour différent MPSoC obtenu avec Synplify

RoC (noeuds)	Tranches	XC2VP100 Slices (%)	BRAM	XC2VP100 BRAM (%)
4	3800	9	77	17
8	9028	20	121	27
16	22554	51	209	47
24	40719	92	297	67
1 ressource	800	2	8	2

La **Figure 5-3** résume par un histogramme le pourcentage d'utilisation du FPGA pour différentes configurations de MPSoC (4, 8, 12, 16, 20 ou 24 ressources). Tel que mentionné ci-haut, on remarque que 20 ressources semblent être une bonne limite d'utilisation pour la puce XC2VP100. Pour 24 ressources, quasiment 100% de la puce serait utilisée. L'histogramme illustre le rapport d'utilisation des tranches par le RoC tel que discuté dans le paragraphe précédent. Le rapport du RoC face au MPSoC est de 18% pour 4 ressources, 29% pour 8 ressources et 38% pour 12 ressources. Avec peu de ressources, le RoC garde un rapport avec l'ensemble du système qu'on peut considérer d'acceptable. Dans un tel cas, optimiser le RoC n'est pas tellement nécessaire, car le rapport d'espace occupé par les ressources est bien plus important. L'optimisation du RoC devient intéressante dans le cas d'un MPSoC à 24 ressources. Réduire l'espace occupé du RoC à 25% pourrait faire passer à 40% son rapport dans le MPSoC.

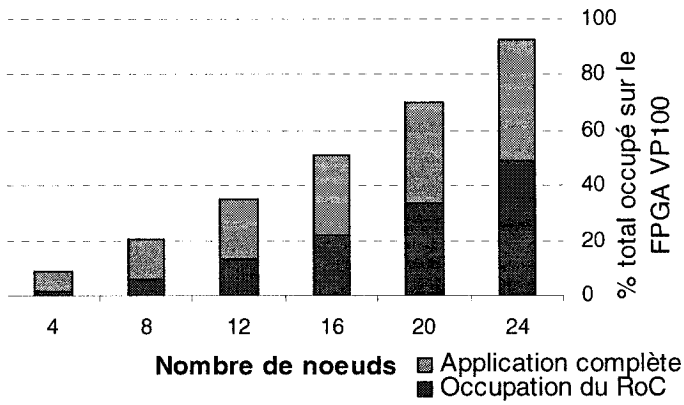


Figure 5-3. Pourcentage d'utilisation du MPSoC pour une puce XC2VP100

De façon similaire, la **Figure 5-4** montre que 8 ressources semblent être la bonne limite d'utilisation pour une puce FPGA XC2VP30 [XILI00]. En effet, 12 ressources utilisent quasiment 100% de la puce. Il est important de savoir que la VP30 possède 3 fois moins de tranches, de FF et de BRAM que la VP100. Même si des optimisations sont apportées au RoC, les ressources utilisent l'ensemble du système. Au bout du compte, la VP30 ne peut pas offrir un environnement suffisamment grand pour tester un RoC avec de nombreuses connexions. La VP30 est donc un environnement de prototypage du RoC (sans application).

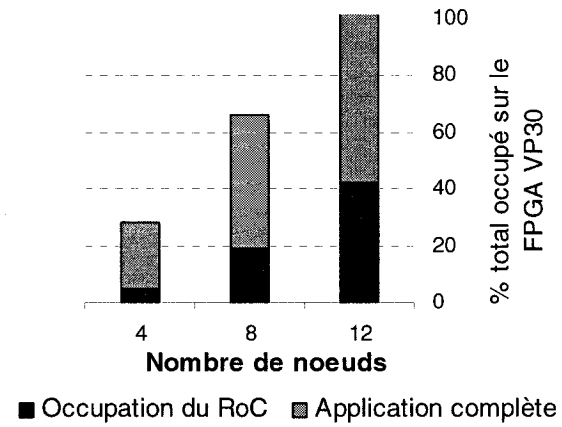


Figure 5-4. Pourcentage d'utilisation du MPSoC pour une puce XC2VP30

5.2.2 Fréquences d'opérations des simulations et de l'implémentation

Malgré le fait que les simulations post-synthèse fonctionnent correctement à 10 MHz, il serait intéressant de confirmer les fréquences de synthèse données par les outils ISE et Synplify [SYNO00]. Pour le moment, nous avons validé un MPSoC à 4 ressources qui fonctionne à 80 MHz. Des fréquences supérieures n'ont pas été testées pour le moment, mais si le système doit dépasser les 100 MHz, une recherche sur la faisabilité de l'implémentation devra être faite.

5.3 Résultats de l'intégration du RoC et du tunnel HyperTransport

L'intégration du RoC avec le tunnel HT a bien fonctionné en simulation. Les spécifications mentionnées dans le chapitre précédent sont vérifiées et ont permis de démontrer que le concept est réalisable. Cette section débute par la présentation des résultats de l'intégration et est suivie d'une discussion sur la complexité du design.

5.3.1 Résultats de l'intégration

La simulation permet de vérifier la faisabilité d'une écriture et d'une lecture vers la mémoire partagée pour des MicroBlaze ayant une interface esclave vers le RoC. Plus précisément, les MicroBlaze ont accès à la mémoire partagée de leur RoC local et de l'autre RoC par l'entremise du tunnel HT. Par un système d'adressage simple, les communications sont transparentes au niveau du processeur. Les adresses sont décodées par l'enveloppe OPB et lorsque celle-ci identifie qu'une requête ne s'adresse pas au RoC local, un paquet est généré pour qu'il soit directement envoyé vers le pont HT. Dans le cas d'une lecture, lorsque le paquet atteint sa destination, l'enveloppe du destinataire génère un nouveau paquet vers le pont HT pour renvoyer la réponse à la source.

Les fonctionnalités du modèle sont limitées mais le but de cette intégration était de fournir une preuve de concept. Cela permet de simplifier l'intégration et d'obtenir les

premiers résultats de latence et de bande passante plus facilement. Le **Tableau 5-3**, résume la latence qu'apporte au système chaque partie du modèle. Il est à noter que ces résultats ne sont pas les mêmes que dans le mémoire d'Ami Castonguay. En effet, le tableau tient compte de récentes améliorations apportées aux machines à états du RoC.

Tableau 5-3. Latence des communications du modèle d'intégration

	Cycles écoulés dans les deux RoCs	Cycles écoulés dans la chaîne HT	Cycles écoulés dans les ponts	Nombre de cycles total
Envoi d'un paquet écriture, DirectRoute activé	18	13	2	33
Envoi d'un paquet écriture, DirectRoute non activé	18	25	2	45
Envoi et réception d'un paquet lecture, DirectRoute activé	36	26	4	66
Envoi et réception d'un paquet lecture, DirectRoute non activé	36	50	4	90

Évidemment, les résultats ne sont pas les mêmes selon qu'il s'agit d'une requête d'écriture ou de lecture. Puisque notre implémentation de l'écriture de requiert pas de confirmation, un paquet d'écriture parcourt en moyenne la moitié des nœuds d'un RoCs. Par contre, un paquet de lecture parcourt tous les nœuds des RoCs pour retourner une réponse. De plus, le paquet de lecture doit parcourir à deux reprises le tunnel HT. Les communications à travers les deux RoCs occupent 40% des cycles. Normalement, chaque fois qu'un paquet est routé par le HT, il doit passer par l'hôte de la chaîne (mentionné à la section 4.4.1). En activant le DirectRoute, les paquets n'ont pas besoin d'effectuer ce chemin et passent donc directement par les deux parties du tunnel HT. La latence des communications diminue dans le tunnel et fait donc passer à 54% le nombre des cycles utilisés par les deux RoCs.

5.3.2 Complexité du modèle

Le modèle intégrant le RoC avec le tunnel HT ne peut pas être synthétisé directement. Comme nous l'avions déjà mentionné, l'hôte du système n'a été développé qu'en logiciel. Pour avoir une idée de la complexité du système par rapport à l'utilisation des ressources matérielles, le **Tableau 5-4** résume les résultats de synthèse obtenus. Ces résultats sont donnés en LUT (Look up tables) au lieu de tranches. Notez qu'une tranche contient 2 LUTs [XILI00]. Les résultats disponibles sur le tunnel HT sont seulement en LUT. Un LUT est l'autre représentation possible de l'utilisation des ressources matérielles d'un FPGA. Les résultats sont obtenus avec l'outil Synplify.

Tableau 5-4. Complexité des composantes faisant partie du modèle

Système	Nombre de LUT pour un FPGA Xilinx
1 Tunnel avec options par défaut	18 100
1 RoC à 4 nœuds, format de 41 bits pour les paquets	823
1 ressource (MicroBlaze, mémoire, bus, contrôleur)	871
Ensemble du modèle : 2 tunnels HT, 2 RoCs et 4 ressources	41 330

Dans le tableau précédent, on observe que le RoC occupe seulement 4% de l'espace du système. Avec sa faible occupation d'espace, le RoC comparé au tunnel permet la communication entre 4 ressources, ce que le tunnel ne peut effectuer tout seul. Le tunnel doit au minimum être connecté à un second tunnel pour être fonctionnel. Avec 41 000 LUTs, l'implémentation du système ne peut pas être effectuée sur une puce VP30 et requiert donc une puce VP100. Même en ajoutant deux ressources à chaque RoC et en agrandissant le RoC pour offrir 6 nœuds (1736 LUTs par RoC), cet ajout ne représenterait que seulement 15% en augmentation par rapport au modèle original. Avant

que la complexité du RoC et des ressources ne fassent le poids face au tunnel, il faudrait un RoC avec de multiples connexions et de ressources. Pour donner une idée, le RoC équivaut en complexité le tunnel lorsqu'il offre 20 connexions (17 439 LUTs par RoC). À petite échelle, le modèle original nous permet de constater que la différence de complexité entre le RoC et le tunnel HT est très grande. Bien évidement, il faut noter que le lien de communication interpuce que permet le tunnel HT est particulier et requiert beaucoup de mécanisme de contrôle. Réaliser une communication interpuce en seulement 13 cycles de latence avec le mode DirectRoute est une fonctionnalité très intéressante du système.

Le RoC ne dicte pas la bande passante maximale des communications. Sa fréquence synthétisée est de 134 MHz comparé à celle du tunnel HT qui est de 103 MHz. Les deux systèmes de communication transmettent leur donnée en 32 bits. Étant donné que le RoC peut gérer des communications non bloquante, théoriquement un paquet peut entrer dans le tunnel à chaque cycle. Les échanges entre chaque RoC par le tunnel HT peuvent donc atteindre un maximum théorique de 412 Mo/s par direction.

5.4 RoC comparé aux autres NoCs

Même si l'architecture du RoC est différente des mailles 2D, il est tout de même possible de comparer le RoC à la maille 2D Lipar [SBKV05]. Les résultats dévoilés dans cet article sont parmi les seuls trouvés dans la littérature qui permettent une comparaison avec l'implémentation du RoC. La complexité des routeurs de Lipar est indépendante de la grosseur de son réseau. Chaque routeur consomme 437 tranches et 478 FFs. Pour une maille 2x2 et 3x3, ils prennent respectivement 1748 (12.76%) et 3934 (28.72%) tranches d'une puce FPGA Xilinx XC2VP30. Le RoC est donc moins volumineux que cette maille 2D jusqu'à concurrence de 8 connexions. Au-delà de 12 connexions, le RoC consomme plus de ressources, mais étant non bloquant, il peut supporter une charge de 100% si tout raccordement entre les sources et les destinations sont une à une. Un autre aspect

intéressant de la comparaison vient du fait que la maille transmet ses paquets par segment d'un FIFO de routeurs à l'autre. Ces transferts s'effectuent sur plusieurs cycles. En plus, lorsque la taille des paquets grandit, cela a pour effet d'augmenter la latence, car les transferts de segments sont plus nombreux. Comparé au RoC, celui-ci transmet ses paquets au complet en un seul cycle d'une banque à l'autre. Pour un RoC à 9 nœuds, la latence est de seulement 23 cycles comparés à 84 cycles pour une maille Lipar 3x3. Cela représente une latence 3,7 fois plus petite. Finalement, il est bon de mentionner que la fréquence d'opération utilisée pour vérifier l'implémentation du RoC est de 80 MHz comparé à celle de la maille qui n'est que de 33 MHz.

Même si la comparaison de certains NoCs avec le RoC diverge sur plusieurs aspects (discuté à la section 1.3), généralement le RoC obtient des performances en bande passante meilleures que certains NoC. Il faut mentionner que le RoC a une architecture implémentée sur FPGA et obtient des vitesses d'opération moindre comparée aux autres NoCs réalisées en ASIC. Le RoC a des caractéristiques avantageuses qui sont propres à lui et qui ne se retrouvent pas dans d'autres architectures. Il permet d'occuper autant d'espace que d'autres NoCs implémentés sur FPGA pour 8 connexions et plus. Citons par exemple, les mailles 3x3 de Hermes [MMMO03] et Lipar vues à la section 1.3 qui consomment autant d'espace que le RoC à 9 connexions. Un commutateur de Hermes consomme 316 tranches. Il faut ainsi 9 fois ce nombre de tranches pour une maille 3x3. Ce qui veut dire 2844 tranches. Pour Lipar, son commutateur coûte 352 tranches, il prend donc 3168 tranches. De son côté, le RoC utilise 3315 tranches pour 9 connexions. Comparé à ceux-ci, le RoC offre moins de latence, plus de bande passante et il fonctionne avec une fréquence d'opération supérieure. En dernière analyse, il ne faut pas oublier de souligner que le RoC effectue ses communications de façon non bloquante tout en ayant une architecture extensible, ce qui n'est pas le cas pour toutes les autres NoCs.

CONCLUSION

Tout au long de ce mémoire, nous avons vu plusieurs aspects du domaine des réseaux intégrés sur puce afin de comprendre de quelle manière est élaborée cette architecture de communication. Nous avons présenté l'implémentation du *Rotator-on-chip*, une architecture de NoC extensible et paramétrable basée sur le modèle réseau de l'anneau par jeton. Le RoC a été vérifié par simulation et par implémentation sur une plateforme de développement pour FPGA. Il a été comparé à une architecture de NoC pour identifier ses points forts et ses points faibles. Ce NoC est une architecture basée sur le modèle d'une maille et ressemble grandement aux autres architectures de NoC. Avec cette comparaison, nous remarquons que pour un RoC à 8 nœuds ou moins, la complexité matérielle du RoC est moindre que l'implémentation d'une maille et que pour 12 nœuds et plus, le RoC consomme davantage de ressources matérielles du FGPA. Cependant, il ne faut pas oublier que le RoC supporte simultanément toutes les communications du réseau si celles-ci sont établies entre une paire de source et de destination unique. En ce qui a trait à la latence des communications, il a été démontré que le RoC obtient une latence 3.7 fois moindre que celle d'un réseau en maille. Le RoC étant extensible, il nous a été possible de l'intégrer dans une plateforme d'exploration pour retirer différentes analyses. Cela a permis de confirmer que le RoC offre des performances intéressantes tout en observant que certaines améliorations sont possibles pour diminuer la latence des communications.

Le RoC est une architecture intéressante, car elle est différente dans sa façon de router les paquets et sa topologie apporte une nouvelle vision au domaine des NoCs qui utilisent trop souvent une topologie en maille. L'architecture du RoC n'est pas parfaite, mais elle offre des avantages indéniables si on la compare à une connexion point à point. Elle permet d'offrir des performances quasi identiques tout en utilisant moins d'espace matériel. Par exemple, l'utilisation de tranches sur un FPGA est inférieure à 25% d'un

VP100 de Xilinx pour la version à 16 nœuds du RoC qui supporte une bande passante agglomérée de 6 Go/sec. Ainsi, tout en offrant une grande bande passante, une partie importante du FPGA est laissée libre à l'élaboration d'un système.

Le RoC est un projet de recherche qui a apporté des contributions sous différents aspects. Nous pouvons citer la nouvelle avenue de recherche qu'a permise l'intégration du RoC au tunnel HyperTransport. Ce projet a montré une façon simple de rendre transparentes les communications à travers des systèmes de communication intra puce et interpuce.

Les travaux de recherche sur le RoC ne se terminent pas ici et nous travaillons présentement à l'intégration de nouvelles fonctionnalités au modèle de bas niveau. Dans le but de réduire l'utilisation des ressources matérielles, nous avons intégré la fonctionnalité d'avoir la moitié moins de canaux de communication. Cette nouvelle architecture du RoC permet d'offrir presque autant de bande passante avec moins de ressources matérielles et obtenir ainsi une complexité matérielle de O(N log N). Cette amélioration est possible par l'introduction d'un nouveau mécanisme de transmission des paquets entre les mémoires tampons des nœuds et les tampons des banques. Lorsqu'un paquet transfert vers la banque, le choix du canal de communication n'est plus en fonction de la destination, mais du canal de communication étant libre. Avant de transférer, si le canal contient un paquet ayant la même destination que le paquet à transmettre, alors le paquet doit attendre au prochain cycle. Cette architecture reste extensible selon les paramètres déjà énoncés. Dans une situation avec un trafic peu élevé, les contraintes sont limitée et permet à l'architecture de résoudre son problème d'augmentation trop grande de sa complexité.

Plusieurs aspects du RoC restent à développer. Les machines à états peuvent être réduites pour ainsi diminuer le nombre de cycles nécessaires pour l'échange des paquets entre les nœuds et les banques. Cet aspect a déjà été optimisé entre la première et la dernière version des machines à états. D'ailleurs, cette dernière optimisation a été intégrée à partir des commentaires provenant de M. Michel Langevin de chez STMicroelectronics. De plus, les parties nœud et banque du RoC sont séparées par l'utilisation de mémoires. Il serait peut-être possible d'utiliser des horloges différentes entre ces parties et ainsi augmenter le débit du réseau. Mais pour tester ces avenues, il est important que le RoC s'équipe d'un bon banc de test pour dénicher les erreurs de conception.

Il sera donc nécessaire de développer une application logicielle qui stressera le réseau. Pour ce faire et pour plusieurs raisons énoncées dans ce qui suit, nous aurions voulu développé un banc de test en langage e. Les principaux avantages de ce banc d'essai en langage *e* seraient :

- La facilité d'interaction avec le code qui permet l'analyse du RoC par la génération de paquets selon différents critères. Il est possible de sauvegarder dans un fichier de rapport le moment de génération d'un paquet et son arrivée à la ressource. Cela permet de vérifier si des paquets se sont perdus dans le réseau
- Le langage e permet l'évaluation du taux de couverture
- Il permet de récolter des données sur le trafic pour identifier les goulots d'étranglement de l'architecture
- Cela permet d'expérimenter une méthodologie nouvelle pour le test et la mise au point d'un NoC

BIBLIOGRAPHIE

[ACGM03] ADRIAHANTENAINA, A., CHARLERY, H., GREINER, A., MORTIEZ, L. 2003. "SPIN: a Scalable, Packet Switched, On-Chip Micro-network", *Design, Automation and Test in Europe Conference and Exhibition*, IEEE, p. 70-73

[ADGR03] ADRIAHANTENAINA, A., GREINER, A. 2003. "Micro-network for SoC: Implementation of a 32-port SPIN network", *Design, Automation and Test in Europe Conference and Exhibition*, IEEE, p. 1128 –1129

[ALNU03] ALHO, M., NURMI, J. 2003. "Implementation of interface router IP for Proteo network-on-chip", *Proceedings of the 6th IEEE International Workshop on Design and Diagnostics of Electronics Circuits and Systems*

[AMIR04] Voir le site Web de Amirix Systems: http://www.amirix.com

[ARM01] ARM. 2001. *AMBA Home Page*. [en ligne]. http://www.arm.com/products/solutions/AMBAHomePage.html (Page consultée le 16 novembre 2006)

[ARTE06] Arteris. 2006. *A comparison of Network on-Chip and Busses*. [en ligne]. http://www.arteris.com/whitepapers.html (Page consultée le 16 novembre 2006)

[BEBE04] BERTOZZI, D., BENINI, L. 2004. "Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-chip", *IEEE circuits and systems magazine*, p. 18-31

[BEDE02] BENINI, L., DE MICHELI, G. 2002. "Networks on Chips: A New Paradigm for Systems on Chip Design", *Design, Automation and Test in Europe Conference and Exhibition*, IEEE, p. 418-419

[BEDE02b] BENINI, L., DE MICHELI, G. 2002. "Networks on Chips: A New SoC Paradigm", *Computer*, IEEE, p. 70-78

[CADE00] Voir le site Web de Cadence : http://www.cadence.com

[CAST06] CASTONGUAY, Ami. 2006. Mémoire. *Plate-forme de communication pour systèmes embarqués multipuces utilisant HyperTransport*. École Polytechnique de Montréal.

[CLEA01] Voir le site Web de Clearspeed™ : http://www.clearspeed.com

[DATO01] DALLY, W., TOWLES, B. 2001. "Route Packets, Not Wires: On-Chip Interconnection Networks", *Design Automation Conference*, IEEE, p. 684-689

[DELI05] DELISLE, K. 2005. *Modélisation d'un réseau intégré sur puce et intégration sur une plate-forme d'exploration architecturale*, Rapport de projet de fin d'études, École Polytechnique de Montréal, 67 p.

[DESL05] DESLAUIERS, François. 2005. Mémoire. *Modélisation d'un réseau intégré sur puce basé sur une architecture en anneau.* École Polytechnique de Montréal.

[FENG81] FENG, T. 1981. "A Survey of Interconnection Networks", *Computer*, IEEE, p. 12-27

[GUGR00] GUERRIER, P., GREINER. A., 2000. "A generic architecture for on-chip packet-switched interconnections", *Design, Automation and Test in Europe Conference and Exhibition*, p. 250-256

[IBM00] Voir le site Web de IBM : http://www.research.ibm.com/cell

[IBM03] IBM. 2003. *CoreConnect™ bus architecture.* [en ligne]. http://www-03.ibm.com/chips/products/coreconnect/ (Page consultée le 24 février 2005)

[JIST00] JIAN, L., SWAMINATHAN, S., TESSIER, R. 2000. "aSOC: A Scalable, Single-Chip communications Architecture", *IEEE International Conference on Parallel Architectures and Compilation Techniques*, p. 37-46

[KAND02] KARIM, F., NGUYEN, A., DEY, S. 2002. "An interconnect architecture for network systems on chips", *Micro*, IEEE, Volume 22, Numéro 5, p. 36-45

[KNDR01] KARIM, F., NGUYEN, A., DEY, S., RAO, R. 2001. "On-chip communication architecture for OC-768 network processors", *38th Design Automation Conference*, p. 678-683

[KJSF02] KUMAR, S., JANTSCH, A., SOININEN, J., FORSELL, M. 2002. "A Network on Chip Architecture and Design Methodology", *IEEE Computer Society Annual Symposium on VLSI,* IEEE. p. 105-112

[MBVV02] MARESCAUX, T., BARTIC, A., VERKEST, D., VERNALDE, S., LAUWEREINS, R. 2002. "Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs", *Field-Programmable Logic and Applications*, p. 795-805

[MENT00] Voir le site Web de Mentor Graphics : http://www.model.com

[MMMO03] MORAES, F., MELLO, A., MOLLER, L., OST, L., CALAZANS, N. 2003. "A Low Area Overhead Packetswitched Network on Chip: Architecture and Prototyping", *IFIP Very Large Scale Integration*

[OCPI00] Voir le site Web de OCP-IP Association : http://www.ocpip.org

[PDGI05] PANDE, P.P., DE MICHELI, G., GRECU, C., IVANOV, A., SALEH, R., 2005. "Design, Synthesis, and Test of Networks on Chips", *IEEE Design & Test of Computers*, numéro 5, volume 22, p. 404-413

[RGRM03] RIJPKEMA, E., GOOSSENS, K., RADULESCU, A., MEERBERGEN, J. 2003. "Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip". *Design Automation and test in Europe*, IEEE, p. 350-355

[SBKV05] SETHURAMAN, B., BHATTACHARYA, P., KHAN, J., VEMURI, R. 2005. " LiPaR: A light-weight parallel router for FPGA-based networks-

on-chip", *Proceedings of the ACM Great Lakes Symposium on VLSI*, p. 452-457

[SINU02] SIGUENZA-TORTOSA, D., NURMI, J. 2002. "Proteo: A New Approach to Network-on-Chip", *IASTED International Conference on Communication Systems and Networks*

[STMI00] Voir le site Web de STMicroelectronics : http://www.st.com

[SYNO00] Voir le site Web de Synopsys : http://www.synopsys.com

[XILI00] Voir le site Web de Xilinx : http://www.xilinx.com

[YOO03] YOO, S. 2003. "On-Chip Communication Design", *DATE Master Course*, p. 67

ANNEXES

ANNEXE A
Fichier logiciel du maître pour tester les communications

Cette section contient le code logiciel exécuté sur le processeur maître. Pour déterminer si l'implémentation du RoC est sans erreur, cette application effectue des tests de lectures et d'écritures sur sa mémoire partagée localement. Cette mémoire est écrite et lu à de nombreuses reprises par les autres processeurs. En communiquant à travers le RoC, ils accèdent cette mémoire et effectue des écrites un après l'autre.

```
include "xparameters.h"
include "xuartns550_l.h"
include "xstatus.h"

define CFG_NS16550_CLK    80000000
define CFG_BAUD          57600
define address1 0x10000000
ddefine addressBRAM 0x0000ffff


void
sendString(const char* pszMessage)
{
while (*pszMessage) {
        if ( *pszMessage == '\n')

XUartNs550_SendByte(XPAR_OPB_UART16550_0_BASEADDR, '\r');
XUartNs550_SendByte(XPAR_OPB_UART16550_0_BASEADDR, *pszMessage++);
        }
}

void
initUart()
{
        /* Set the baud rate and number of stop bits */
        XUartNs550_SetBaud(        XPAR_OPB_UART16550_0_BASEADDR,
                                        CFG_NS16550_CLK,
                                        CFG_BAUD);
   XUartNs550_mSetLineControlReg(
        XPAR_OPB_UART16550_0_BASEADDR,
```

```
        XUN_LCR_8_DATA_BITS);
    /* Enable the FIFOs for 16550 mode since the device defaults to no FIFOs */
    XUartNs550_mWriteReg( XPAR_OPB_UART16550_0_BASEADDR,
                                                    XUN_FCR_OFFSET,
                                                    XUN_FIFO_ENABLE);
}

void convertIntegerToString(char* pszBuffer, int iValue)
{
        sprintf(pszBuffer, "%d", iValue);
}


int
main(int argc, char *argv[])
{
        int* test_mem = address1;
        char pszBuffer[33];
        int bob;
        int i;

        initUart();

        sendString("Début du test écriture de 1! \n\n");
        * test_mem = 1;

        sendString("WHILE(1)\n\n");

        while(1)
        {
                        if (*test_mem == 2)
                        {

                                sendString("Esclave 0 ok! =");
                                convertIntegerToString(pszBuffer, * test_mem);
                                sendString(pszBuffer);
                                sendString("\n\n");
                                * test_mem = 3;

                        }

                        if (*test_mem == 3)
                        {
```

```
        sendString("test 3  = ");
        convertIntegerToString(pszBuffer, * test_mem);
        sendString(pszBuffer);
        sendString("\n\n");

}

if (*test_mem == 4)
{

        sendString("test 4  = ");
        convertIntegerToString(pszBuffer, * test_mem);
        sendString(pszBuffer);
        sendString("\n\n");

}

if (*test_mem == 5)
{

        sendString("test 5 = ");
        convertIntegerToString(pszBuffer, * test_mem);
        sendString(pszBuffer);
        sendString("\n\n");

}

if (*test_mem == 6)
{

        sendString("test 6 = ");
        convertIntegerToString(pszBuffer, * test_mem);
        sendString(pszBuffer);
        sendString("\n\n");

}

if (*test_mem == 7)
{

        sendString("test 7 = ");
        convertIntegerToString(pszBuffer, * test_mem);
```

```
                    sendString(pszBuffer);
                    sendString("\n\n");

                }

                if (*test_mem == 8)
                {

                    sendString("Esclave 1, 2, 3 et le maître sont ok! =");
                    convertIntegerToString(pszBuffer, * test_mem);
                    sendString(pszBuffer);
                    sendString("\n\n");
                    sendString("Un dernier test! \n\n");
                    * test_mem = 9;

                }

                if (*test_mem == 9)
                {

                    sendString("A la fin test_mem vaut = ");
                    convertIntegerToString(pszBuffer, * test_mem);
                    sendString(pszBuffer);
                    sendString("\n\n");
                    sendString("Le test est fini! \n\n");
                    break;
                }
        }
        return ;
}
```

ANNEXE B
Fichier package de configuration du RoC

Ce fichier permet de jouer sur les différents paramètres de configuration du RoC. Le RoC étant générique, changer une valeur parmi les prochaines change automatiquement l'ensemble de l'architecture. Tous les autres fichiers pour le RoC dépendent de ce package pour la déclaration des signaux, les tailles et itérations pour la génération des composantes interne du RoC.

```
library ieee;
use ieee.std_logic_1164.all;

package roc_package is
        constant ROC_TRIANGLE_NB : integer := 4;
        constant ROC_BANK_NB   : integer := 4;

        constant ROC_ADDR_SIZE : integer := 4;--attention relie avec Triangle_NB
        constant PLAGE_SIZE        : integer := 4;--attention relie avec Triangle_NB

        constant ROC_DATA_SIZE : integer := 32;
        constant ROC_USER_SIZE  : integer := 1;

        constant NB_OF_MASTER  : integer := 1;
        constant NB_OF_SLAVE     : integer := ROC_TRIANGLE_NB-
NB_OF_MASTER;

        constant ROC_MESG_SIZE : integer :=
ROC_ADDR_SIZE+ROC_ADDR_SIZE+ROC_DATA_SIZE+ROC_USER_SIZE;

        --/Source/Destination/Message/User-bits/
        --constant SLAVE_0_C_BASEADDR = --0x00000000
        --constant SLAVE_0_C_HIGHADDR = --0x0000ffff
        --configuration de l'adresse des resssources slave

        subtype roc_message_typ is std_logic_vector(ROC_MESG_SIZE-1 downto 0);
        subtype plage_typ is std_logic_vector(PLAGE_SIZE-1 downto 0);
        subtype opb_addr_bus_typ is std_logic_vector(ROC_ADDR_SIZE-1 downto 0);
        subtype opb_data_bus_typ is std_logic_vector(ROC_DATA_SIZE-1 downto 0);
        subtype vectorBE is std_logic_vector(3 downto 0);

type roc_message_lst is array (0 to ROC_TRIANGLE_NB-1) of roc_message_typ;
```

```
type rotate_lst         is array (0 to ROC_TRIANGLE_NB-1) of roc_message_lst;
type roc_bank_lst               is array (0 to ROC_BANK_NB-1) of roc_message_lst;

--fils inter-bangque
type roc_plage_lst      is array (0 to ROC_TRIANGLE_NB-1) of plage_typ;
type roc_bit_lst                is array (0 to ROC_TRIANGLE_NB-1) of std_logic;
type bitmap_lst                 is array (0 to ROC_TRIANGLE_NB-1) of roc_bit_lst;
type banque_info_lst is array (0 to ROC_BANK_NB-1) of std_logic;

type wrap_master_lst is array (0 to NB_OF_MASTER-1) of std_logic;
type wrap_slave_lst   is array (0 to NB_OF_SLAVE-1) of std_logic;
type wrap_slaveBE_lst is array (0 to NB_OF_SLAVE-1) of vectorBE;

type    ADDRs_banque_lst is array (0 to ROC_BANK_NB-1) of opb_addr_bus_typ;
type    opb_ABus_M_lst is array (0 to NB_OF_MASTER-1) of opb_addr_bus_typ;
type    opb_ABus_S_lst is array (0 to NB_OF_SLAVE-1) of opb_addr_bus_typ;
type    opb_DBus_lst is array (0 to ROC_TRIANGLE_NB-1) of opb_data_bus_typ;
type    opb_DBus_M_lst is array (0 to NB_OF_MASTER-1) of opb_data_bus_typ;
type    opb_DBus_S_lst is array (0 to NB_OF_SLAVE-1) of opb_data_bus_typ;

----------pour top avec ROC + EDK
subtype vector32_to is std_logic_vector(0 to 31);
subtype vector32_downto is std_logic_vector(31 downto 0);
subtype vector3_to is std_logic_vector(0 to 3);
subtype vector3_downto is std_logic_vector(3 downto 0);
type vec32bits_master_to is array (0 to NB_OF_MASTER-1) of vector32_to;
type vec32bits_slave_to is array (0 to NB_OF_SLAVE-1) of vector32_to;
type vec32bits_master_downto is array (0 to NB_OF_MASTER-1) of
vector32_downto;
type vec32bits_slave_downto is array (0 to NB_OF_SLAVE-1) of vector32_downto;
type vec4bits_slave_to is array (0 to NB_OF_SLAVE-1) of vector3_to;
type vec4bits_master_to is array (0 to NB_OF_MASTER-1) of vector3_to;
type vec4bits_slave_downto is array (0 to NB_OF_SLAVE-1) of vector3_downto;

end roc_package;
```

ANNEXE C
Fichier *top level*

Ce fichier réplique les banques et les nœuds du RoC autant que voulue et les interconnecte pour former l'ensemble du réseau RoC.

```
library ieee;
library work;
use work.roc_package.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity ROC_TOP_GEN is
port (CLK,RESET         :in std_logic;
      WR_EN             :in roc_bit_lst;
      WR_CLK            :in roc_bit_lst;
      RD_EN             :in roc_bit_lst;
      RD_CLK            :in roc_bit_lst;
      DIN               :in roc_message_lst;
      FULL              :out  roc_bit_lst;
      EMPTY             :out  roc_bit_lst;
      WR_ACK            :out  roc_bit_lst;
      RD_ACK            :out  roc_bit_lst;
      WR_ERR            :out  roc_bit_lst;
      RD_ERR            :out  roc_bit_lst;
      DOUT              :out roc_message_lst);
end ROC_TOP_GEN;

architecture ROC_TOP_GEN_arch of ROC_TOP_GEN is

component ROC is
generic(plage_adresse : integer);
port (CLK,RESET                 :in   std_logic;
      WR_EN,WR_CLK              :in   std_logic;
      RD_EN,RD_CLK              :in   std_logic;
      DIN                       :in   roc_message_typ;
      IN_ROC                    :in   roc_message_lst;
            FULL                :out  std_logic;
            EMPTY               :out  std_logic;
            WR_ACK              :out  std_logic;
            RD_ACK              :out  std_logic;
            WR_ERR              :out  std_logic;
```

```
                RD_ERR              :out    std_logic;
                DOUT                :out    roc_message_typ;
                OUT_ROC             :out    roc_message_lst;
                bitmap_banques_in   :in     roc_bit_lst;
                bitmap_banques_out  :out    roc_bit_lst);
end component;

signal rotate   :rotate_lst;
signal bitmap   :bitmap_lst;

begin

gen_roc_triangle : for roc_itr in 1 to ROC_TRIANGLE_NB-2 generate
roc_inst_normal : roc
generic map(plage_adresse => roc_itr)
port map (CLK,RESET,WR_EN(roc_itr),WR_CLK(roc_itr),RD_EN(roc_itr),
        RD_CLK(roc_itr),DIN(roc_itr),rotate(roc_itr-1),FULL(roc_itr),
        EMPTY(roc_itr),WR_ACK(roc_itr),RD_ACK(roc_itr),WR_ERR(roc_itr),RD_
        ERR(roc_itr),DOUT(roc_itr),rotate(roc_itr),bitmap(roc_itr-1),bitmap(roc_itr));
end generate gen_roc_triangle;

roc_inst_debut : roc
generic map(plage_adresse => 0)
port map (CLK,RESET,WR_EN(0),WR_CLK(0),RD_EN(0),RD_CLK(0),DIN(0),
        rotate(ROC_TRIANGLE_NB-1),FULL(0),EMPTY(0),WR_ACK(0),
        RD_ACK(0),WR_ERR(0),RD_ERR(0),DOUT(0),rotate(0),bitmap(ROC_TRIA
        NGLE_NB-1),bitmap(0));

roc_inst_fin : roc
generic map(plage_adresse => ROC_TRIANGLE_NB-1)
port map (CLK,RESET,WR_EN(ROC_TRIANGLE_NB-1),
        WR_CLK(ROC_TRIANGLE_NB-1),RD_EN(ROC_TRIANGLE_NB-1),
        RD_CLK(ROC_TRIANGLE_NB-1),DIN(ROC_TRIANGLE_NB-1),
        rotate(ROC_TRIANGLE_NB-2),FULL(ROC_TRIANGLE_NB-1),
        EMPTY(ROC_TRIANGLE_NB-1),WR_ACK(ROC_TRIANGLE_NB-1),
        RD_ACK(ROC_TRIANGLE_NB-1),WR_ERR(ROC_TRIANGLE_NB-1),
        RD_ERR(ROC_TRIANGLE_NB-1),DOUT(ROC_TRIANGLE_NB-1),
        rotate(ROC_TRIANGLE_NB-1),bitmap(ROC_TRIANGLE_NB-2),
        bitmap(ROC_TRIANGLE_NB-1));
end ROC_TOP_GEN_arch;
```

ANNEXE D
Fichier du nœud et la banque

Ce fichier connecte chacune des composantes du RoC. Le *top level* permet de connecter ce qui est en fait plusieurs tranches du RoC. Ces tranches sont nommées RoC (RoC top générique) et sont des combinaisons nœud + banque. Le code présent connecte chacune des composantes du nœud et de la banque entre elles pour former le RoC. Finalement, ce fichier est répliqué plusieurs fois dans le *top level* pour former l'ensemble du réseau.

```
library ieee;
library work;
use work.roc_package.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ROC is
generic(plage_adresse : integer :=3);
port (CLK,RESET          :in    std_logic;
      WR_EN,WR_CLK       :in    std_logic;
      RD_EN,RD_CLK       :in    std_logic;
      DIN                :in    roc_message_typ;
      IN_ROC             :in    roc_message_lst;
      FULL               :out   std_logic;
      EMPTY              :out   std_logic;
      WR_ACK             :out   std_logic;
      RD_ACK             :out   std_logic;
      WR_ERR             :out   std_logic;
      RD_ERR             :out   std_logic;
      DOUT               :out   roc_message_typ;
      OUT_ROC            :out   roc_message_lst;
      bitmap_banques_in  :in    roc_bit_lst;
      bitmap_banques_out:out    roc_bit_lst);
end ROC;

architecture ROC_arch of ROC is

component fifo_to_reg is
port (CLK,RESET,ACK    :in std_logic;
      ADDR             :in std_logic_vector(ROC_ADDR_SIZE-1 downto 0);
      INFO_TAMPON      :in roc_bit_lst;
```

```
        ENABLE              :out std_logic;
        set_reg             :out roc_bit_lst;
        INFO_TAMPON_ENTRE :out std_logic_vector(0 to ROC_TRIANGLE_NB-
1));
end component;

component controle_noyau
generic(plage_adresse : integer);
port (CLK,RESET             :in     std_logic;
        info_tampon         :in     roc_bit_lst;
        bitmap_banques_in   :in     roc_bit_lst;
        set_muxs            :out    roc_bit_lst;
        set_banques         :out    banque_info_lst;
        set_reg_sortie      :out    std_logic;
        clr_bitmap_regs     :out    roc_bit_lst;
        bitmap_banques_out  :out    roc_bit_lst;
        bitmap_dout_entre   :in     std_logic);
end component;

component fifo_roc
port (din               :in     roc_message_typ;
        wr_en           :in     std_logic;
        wr_clk          :in     std_logic;
        rd_en           :in     std_logic;
        rd_clk          :in     std_logic;
        ainit           :in     std_logic;
        dout            :out    roc_message_typ;
        full            :out    std_logic;
        empty           :out    std_logic;
        rd_ack          :out    std_logic;
        rd_err          :out    std_logic;
        wr_ack          :out    std_logic;
        wr_err          :out    std_logic);
end component;

component info_tampons_entre is
port (clk,reset             :in     std_logic;
        info_tampon_noyau   :in     std_logic;
        info_tampon_entre   :in     std_logic;
        info_tampon         :out    std_logic);
end component;

component banque
```

```
port (D      :in    roc_message_typ;
      Q      :out   roc_message_typ;
      CLK    :in    std_logic;
      CE     :in    std_logic;
      ACLR   :in    std_logic);
end component banque;


component mux_2_1
port (A      :in    roc_message_typ;
      B      :in    roc_message_typ;
      SEL    :in    std_logic;
      Y      :out   roc_message_typ);
end component mux_2_1;

--config pour la synthèse
attribute syn_black_box : boolean;
attribute syn_black_box of fifo_roc: component is true;
attribute syn_black_box of banque: component is true;

signal enable_entre,ack_entre,wr_err_sortie,ack_sortie,rd_err_entre,
      full_sortie,empty_entre,set_reg_sortie,bitmap_dout_entre : std_logic;
signal dout_entre      : roc_message_typ;
signal set_reg,info_tampon,clr_bitmap_regs,set_muxs : roc_bit_lst;
signal info_tampon_entre : std_logic_vector(0 to ROC_TRIANGLE_NB-1);
signal set_banques : banque_info_lst;
signal in_banque,out_banque,signal_out_reg : roc_message_lst;

begin

fifo_entre: fifo_roc
  port map (din,wr_en,wr_clk,enable_entre,rd_clk,reset,dout_entre,full,
                    empty_entre,ack_entre,rd_err_entre,wr_ack,wr_err);
fifo_sortie: fifo_roc
  port map (IN_ROC(plage_adresse),set_reg_sortie,wr_clk,rd_en,
      rd_clk,reset,dout,full_sortie,empty,rd_ack,rd_err,ack_sortie,wr_err_sortie);

gen_info_tampons_entre : for i in 0 to ROC_TRIANGLE_NB-1 generate
if_gen_reg_A:
if plage_adresse /= 0 and i /= plage_adresse and i /= plage_adresse-1 generate
      info_tampons_entre_1: info_tampons_entre
      port map ( clk,reset,clr_bitmap_regs(i),info_tampon_entre(i),info_tampon(i));
      end generate if_gen_reg_A;
```

```
if_gen_reg_B: if plage_adresse = 0 and i /= plage_adresse and i /=
ROC_TRIANGLE_NB-1 generate
        info_tampons_entre_1: info_tampons_entre
                port map ( clk,reset,clr_bitmap_regs(i),info_tampon_entre(i),
                info_tampon(i));
        end generate if_gen_reg_B;
end generate gen_info_tampons_entre;


gen_reg : for i in 0 to ROC_TRIANGLE_NB-1 generate
if_gen_reg_A:
if plage_adresse /= 0 and i /= plage_adresse and i /= plage_adresse-1 generate
        REG_interne: banque
        port map (dout_entre,signal_out_reg(i),CLK,set_reg(i),RESET);
        end generate if_gen_reg_A;

if_gen_reg_B:
if plage_adresse = 0 and i /= plage_adresse and i /= ROC_TRIANGLE_NB-1 generate
        REG_interne: banque
        port map (dout_entre,signal_out_reg(i),CLK,set_reg(i),RESET);
        end generate if_gen_reg_B;
end generate gen_reg;


gen_mux_2_1 : for i in 0 to ROC_TRIANGLE_NB-1 generate
if_mux_2_1_A:
if plage_adresse /= 0 and i /= plage_adresse and i /= plage_adresse-1 generate
        mux_reg: mux_2_1
        port map (signal_out_reg(i),IN_ROC(i),set_muxs(i),in_banque(i));
        end generate if_mux_2_1_A;

if_mux_2_1_B:
if plage_adresse = 0 and i /= plage_adresse and i /= ROC_TRIANGLE_NB-1 generate
        mux_reg: mux_2_1
        port map (signal_out_reg(i),IN_ROC(i),set_muxs(i),in_banque(i));
        end generate if_mux_2_1_B;
        end generate gen_mux_2_1;

gen_banque : for i in 0 to ROC_TRIANGLE_NB-1 generate
if_gen_banque_A:
if plage_adresse /= 0 and i /= plage_adresse and i /= plage_adresse-1 generate
        REG_banque: banque
```

```
        port map (in_banque(i),out_banque(i),CLK,set_banques(i),RESET);
        end generate if_gen_banque_A;

if_gen_banque_B:
if plage_adresse = 0 and i /= plage_adresse and i /= ROC_TRIANGLE_NB-1 generate
        REG_banque: banque
        port map (in_banque(i),out_banque(i),CLK,set_banques(i),RESET);
        end generate if_gen_banque_B;
end generate gen_banque;

gen_zero : for i in 0 to ROC_TRIANGLE_NB-1 generate
if_gen_zero_A: if plage_adresse = 0 and i = ROC_TRIANGLE_NB-1 generate
        info_tampon(i)<='0';
        info_tampon(0)<='0';
        end generate if_gen_zero_A;
if_gen_zero_B: if plage_adresse /= 0 and i = plage_adresse generate
        info_tampon(i)<='0';
        info_tampon(i-1)<='0';
        end generate if_gen_zero_B;
end generate gen_zero;

gen_OUT_ROC : for i in 0 to ROC_TRIANGLE_NB-1 generate
if_gen_OUT_ROC_debut_A:
if plage_adresse = 0 and i /= 0 and i /= ROC_TRIANGLE_NB-1 generate
        OUT_ROC(i)<=out_banque(i);
        end generate if_gen_OUT_ROC_debut_A;
if_gen_OUT_ROC_debut_B:
if plage_adresse = 0 and i = ROC_TRIANGLE_NB-1 generate
        OUT_ROC(i)<=dout_entre;
        bitmap_dout_entre<=INFO_TAMPON_ENTRE(i);
        end generate if_gen_OUT_ROC_debut_B;

if_gen_OUT_ROC_NB_A:
if plage_adresse /= 0 and i = plage_adresse-1 generate
        OUT_ROC(i)<=dout_entre;
        bitmap_dout_entre<=INFO_TAMPON_ENTRE(i);
        end generate if_gen_OUT_ROC_NB_A;
if_gen_OUT_ROC_NB_B:
if plage_adresse /= 0 and i /= plage_adresse and i /= plage_adresse-1 generate
        OUT_ROC(i)<=out_banque(i);
        end generate if_gen_OUT_ROC_NB_B;
end generate gen_OUT_ROC;
```

```
fifo_to_reg_1: fifo_to_reg
        port map (CLK,RESET,ACK_entre,
DOUT_entre(ROC_MESG_SIZE-ROC_ADDR_SIZE-1 downto ROC_MESG_SIZE-
ROC_ADDR_SIZE-ROC_ADDR_SIZE),INFO_TAMPON,
ENABLE_entre,set_reg,INFO_TAMPON_ENTRE);

ctrl_noy: controle_noyau
generic map(plage_adresse => plage_adresse)
        port map (CLK,RESET,info_tampon,bitmap_banques_in,set_muxs,
        set_banques,set_reg_sortie,clr_bitmap_regs,
bitmap_banques_out,bitmap_dout_entre);

end ROC_arch;
```

ANNEXE E
Machine à état du nœud

```
library IEEE;
library work;
use IEEE.std_logic_1164.all;
use work.roc_package.all;
use ieee.std_logic_unsigned.all;

entity fifo_to_reg is
port (CLK,RESET,ACK       :in std_logic;
      ADDR                :in std_logic_vector(ROC_ADDR_SIZE-1 downto 0);
      INFO_TAMPON         :in roc_bit_lst;
      ENABLE              :out std_logic;
      set_reg             :out roc_bit_lst;
      INFO_TAMPON_ENTRE :out std_logic_vector(0 to ROC_TRIANGLE_NB-1)
      );
end fifo_to_reg;

architecture fifo_to_reg_arch of fifo_to_reg is

type state is (e1,e2);
signal etat,etat_suiv : state;

begin
process (clk,reset)
begin
if reset = '1' then
      etat <= e1;
elsif clk'event and clk ='0' then
      etat <= etat_suiv;
end if;
end process;

process(etat,ACK,ADDR,INFO_TAMPON)
begin
enable<='1';
set_reg<=(others=>'0');
INFO_TAMPON_ENTRE<=(others=>'0');

case etat is

when e1 =>    if ACK = '1' and INFO_TAMPON(conv_integer(ADDR)-1)='0' then
```

```
                    enable<='1';
                    if conv_integer(ADDR) /= 0 then
                        set_reg(conv_integer(ADDR)-1) <= '1';
                        INFO_TAMPON_ENTRE(conv_integer(ADDR)-1) <=
'1';
                    end if;
                    etat_suiv<=e1;
                elsif ACK = '1' and INFO_TAMPON(conv_integer(ADDR)-1)='1' then
                    enable<='0';
                    if conv_integer(ADDR) /= 0 then
                        set_reg(conv_integer(ADDR)-1) <= '0';
                        INFO_TAMPON_ENTRE(conv_integer(ADDR)-1) <=
'0';
                    end if;
                    etat_suiv<=e2;
                else    enable<='1';
                    if conv_integer(ADDR) /= 0 then
                        set_reg(conv_integer(ADDR)-1) <= '0';
                        INFO_TAMPON_ENTRE(conv_integer(ADDR)-1) <=
'0';
                    end if;
                    etat_suiv<=e1;
                end if;

when e2 =>      if INFO_TAMPON(conv_integer(ADDR)-1)='0' then
                    enable<='1';
                    if conv_integer(ADDR) /= 0 then
                        set_reg(conv_integer(ADDR)-1) <= '1';
                        INFO_TAMPON_ENTRE(conv_integer(ADDR)-1) <=
'1';
                    end if;
                    etat_suiv<=e1;
                else    enable<='0';
                    if conv_integer(ADDR) /= 0 then
                        set_reg(conv_integer(ADDR)-1) <= '0';
                        INFO_TAMPON_ENTRE(conv_integer(ADDR)-1) <=
'0';
                    end if;
                    etat_suiv<=e2;
                end if;

when others =>      enable<='0';
                if conv_integer(ADDR) /= 0 then
```

```
                    set_reg(conv_integer(ADDR)-1) <= '0';
                    INFO_TAMPON_ENTRE(conv_integer(ADDR)-1) <= '0';
                end if;
                etat_suiv<=e1;
end case;
end process;
end fifo_to_reg_arch;
```

ANNEXE F
Machine à état de la banque

```
library IEEE;
library work;
use IEEE.std_logic_1164.all;
use work.roc_package.all;
use ieee.std_logic_unsigned.all;

entity controle_noyau is
generic(plage_adresse : integer := 3);
port (CLK,RESET                 :in     std_logic;
      info_tampon               :in     roc_bit_lst;
      bitmap_banques_in         :in     roc_bit_lst;
      set_muxs                  :out    roc_bit_lst;
      set_banques               :out    banque_info_lst;
      set_reg_sortie            :out    std_logic;
      clr_bitmap_regs           :out    roc_bit_lst;
      bitmap_banques_out        :out    roc_bit_lst;
      bitmap_dout_entre         :in     std_logic);
end controle_noyau;

architecture controle_noyau_arch of controle_noyau is
begin


set_muxs<=bitmap_banques_in;

gen_clr: for i in 0 to ROC_TRIANGLE_NB-1 generate
      clr_bitmap_regs(i)<=info_tampon(i) and not bitmap_banques_in(i);
      set_banques(i)<=info_tampon(i) or bitmap_banques_in(i);
end generate gen_clr;

if_gen0_proc: if plage_adresse = 0 generate
process(clk,reset,info_tampon,bitmap_banques_in,bitmap_dout_entre)
begin
if reset = '1' then
  set_reg_sortie<='0';
      bitmap_banques_out<=(others=>'0');
elsif clk'event and clk='1' then
      gen_for: for i in 0 to ROC_TRIANGLE_NB-1 loop
            if_gen: if i /= plage_adresse and i /= ROC_TRIANGLE_NB-1 then
                  bitmap_banques_out(i) <=info_tampon(i) or
```

```
bitmap_banques_in(i);
                elsif i = ROC_TRIANGLE_NB-1 then
                        bitmap_banques_out(i) <=bitmap_dout_entre;
                else    bitmap_banques_out(i) <='0';
                end if if_gen;
        end loop gen_for;
        set_reg_sortie <= bitmap_banques_in(plage_adresse);
end if;
end process;
end generate if_gen0_proc;

if_gen_proc: if plage_adresse /= 0 generate
process(clk,reset,info_tampon,bitmap_banques_in,bitmap_dout_entre)
begin
if reset = '1' then
  set_reg_sortie<='0';
        bitmap_banques_out<=(others=>'0');
elsif clk'event and clk='1' then
        gen_for: for i in 0 to ROC_TRIANGLE_NB-1 loop
                if_gen: if i /= plage_adresse and i /= plage_adresse-1 then
                        bitmap_banques_out(i) <=info_tampon(i) or
bitmap_banques_in(i);
                elsif i = plage_adresse-1 then
                        bitmap_banques_out(i) <=bitmap_dout_entre;
                else    bitmap_banques_out(i) <='0';
                end if if_gen;
        end loop gen_for;
        set_reg_sortie <= bitmap_banques_in(plage_adresse);
end if;
end process;
end generate if_gen_proc;

end controle_noyau_arch;
```

ANNEXE G
Ficher du multiplexeur pour le tampon de banque

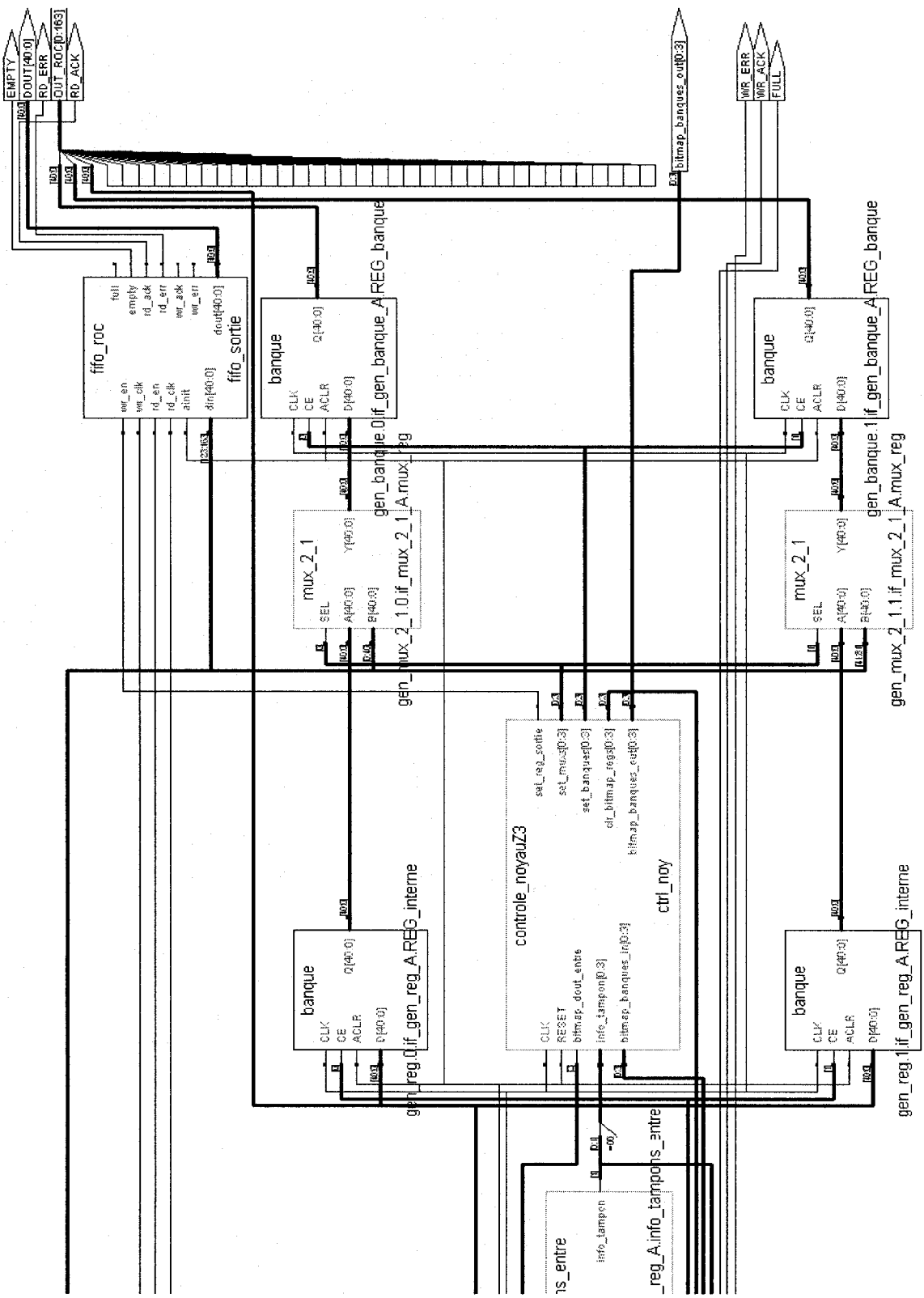
```
library ieee;
library work;
use work.roc_package.all;
use ieee.std_logic_1164.all;

ENTITY mux_2_1 is
PORT(  A,B: IN roc_message_typ;
          SEL: IN STD_LOGIC;
          Y: OUT roc_message_typ);
END mux_2_1;

architecture mux_2_1_arch of mux_2_1 is

begin
process(SEL,A,B)
begin

case SEL is
        when '0' => Y <= A;
        when '1' => Y <= B;
        when others => Y <= (others => 'U');
end case;
end process;
end mux_2_1_arch;
```

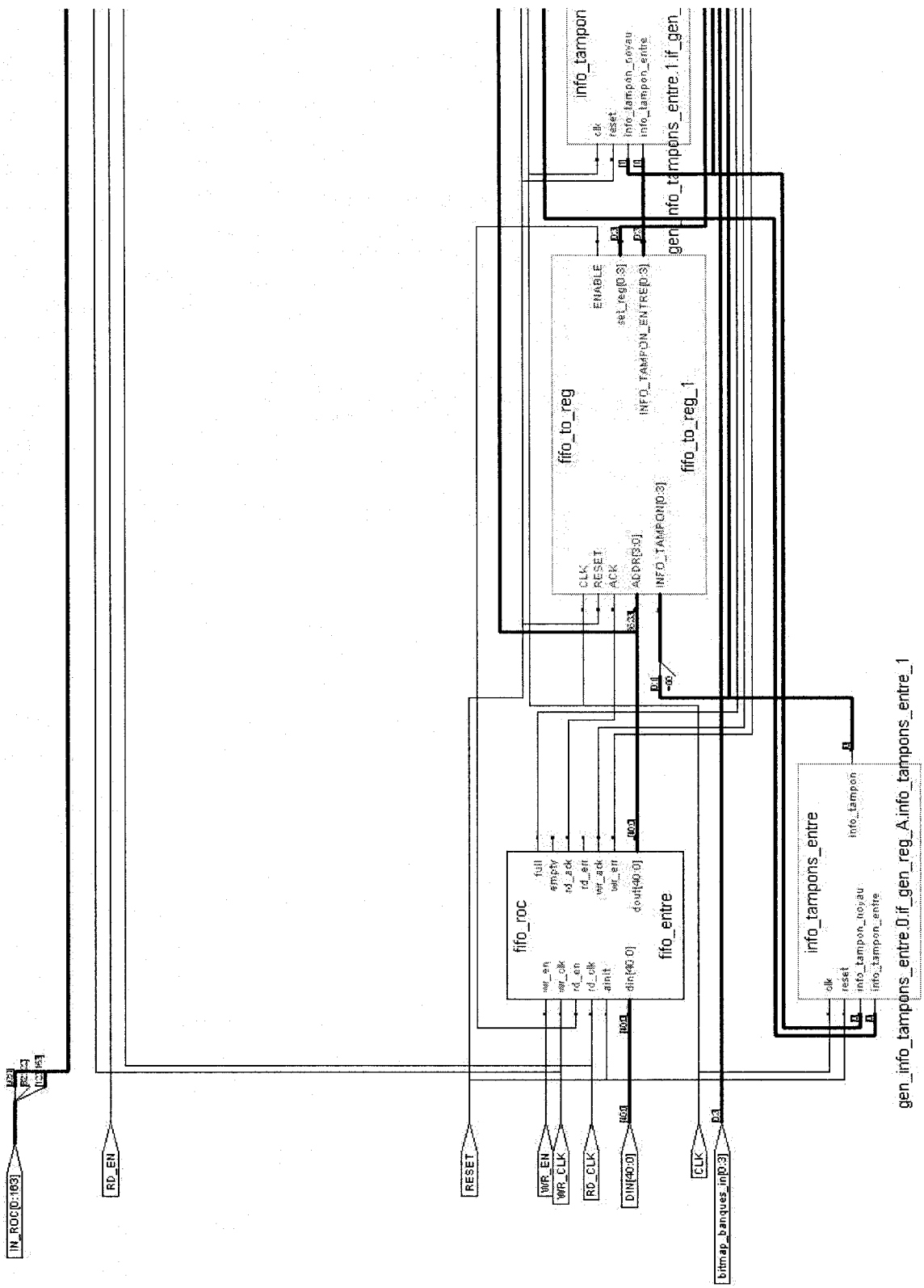
ANNEXE H
Ficher d'une mémoire *bitmap*

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.roc_package.all;
use ieee.std_logic_unsigned.all;

entity info_tampons_entre is
port (clk,reset                 :in std_logic;
        info_tampon_noyau       :in std_logic;
        info_tampon_entre       :in std_logic;
        info_tampon             :out std_logic);
end info_tampons_entre;

architecture info_tampons_entre_arch of info_tampons_entre is

begin
process(clk,reset)--info_tampon_noyau,info_tampon_entre,RESET)
begin
if reset='1' then--or info_tampon_noyau='1' then
        info_tampon<='0';
elsif clk'event and clk='1' then
        if info_tampon_noyau='1' then
                info_tampon<='0';
        elsif info_tampon_entre='1' then
                info_tampon<='1';
        end if;
end if;
end process;
end info_tampons_entre_arch;
```

ANNEXE I
Schéma du RoC

Les deux prochaines pages son un copier collé du schéma généré par l'outil Synplify pour l'ensemble des composantes d'une tranche de RoC. Le schéma est divisé en deux parties, de la même façon que l'on divise le RoC en une banque et un nœud. Le premier schéma présente les composantes du nœud à l'exception de deux banques (REG_interne) appartenant au nœud. Les quatre composantes nommées « banque » sont en fait les éléments de mémoire tampon du nœud et de la banque. Pour accommoder la présentation du schéma, la machine à état du nœud est séparée sur deux pages.


EMPTY
DOUT[40:0]
RD_ERR
OUT_ROC[0:163]
RD_ACK
WR_ERR
WR_ACK
FULL
bitmap_banques_out[0:3]
fifo_roc
fifo_sortie
banque
mux_2_1
controle_noyauZ3
ctrl_noy
gen_banque.0if_gen_banque_A.REG_banque
gen_banque.1if_gen_banque_A.REG_banque
gen_mux_2_1_0.if_mux_2_1_A.mux_reg
gen_mux_2_1_1.if_mux_2_1_A.mux_reg
gen_reg.0.if_gen_reg_A.REG_interne
gen_reg.1.if_gen_reg_A.REG_interne
info_tampon
_reg_A.info_tampons_entre


IN_ROC[0:163]
RD_EN
RESET
WR_EN
WR_CLK
RD_CLK
DIN[40:0]
CLK
bitmap_banques_in[0:3]
fifo_roc
fifo_entre
fifo_to_reg
fifo_to_reg_1
info_tampons_entre
info_tampon
gen_info_tampons_entre.0.if_gen_reg_A.info_tampons_entre_1